# Manycore Operating Systems for Safety-Critical Systems

Der Fakultät für Angewandte Informatik
der Universität Augsburg
zur Erlangung der Lehrbefähigung im Lehrgebiet Informatik
vorgelegte Habilitationsschrift

von
Dr. rer. nat. Florian Kluge

geboren am 19. August 1979
in Donauwörth

Januar 2016

# Abstract

Technological advancements enable to integrate more and more processing cores on single chips. After several years of multicore processors, in the last years the first manycore processors with 64 and more cores have reached the markets. Concurrently, designers of safety-critical systems strive to integrate more and more powerful software in their systems. For example, advanced driver assistence system increase travelling comfort, but can also improve car safety. Manycore processors can deliver the performance needed by such applications. Due to special requirements in safety-critical systems, a direct use of these processors is mostly hindered. To make them usable in safety-critical domains, existing concepts for software design need to be rethought and new concepts need to be developed. The operating system plays a key role in this process, as it provides the "glue" between application software and hardware platform.

This work investigates, how future operating systems for manycore processors should be designed such that they can be deployed in safety-critical systems. A manycore operating system for safety-critical applications (MOSSCA) is designed and applied to several use-cases. Operating system functionalities of MOSSCA are distributed over the cores of a manycore processor as far as possible. MOSSCA provides means to develop applications accordingly. Also it provides the platform for further investigations of operating system mechanisms. One of these is a timing analysis of the boot process in a manycore processor. Further considerations on shared resources show that the timing behaviour of applications is often abstracted too far in scheduling models, thus prohibiting optimisations or the exploitation of existing tolerances. A generic timing model (GTM) is developed to capture timing properties and requirements in cyber-physical system (CPS) during their development. One outcome of GTM are history-cognisant utility functions that can be applied for scheduling. In this work, their ability to map the constraints of $(m, k)$-firm real-time tasks is examined more closely. Beyond these, a number of further aspects is still being investigated, for example the coordination between tasks in a manycore processor and the further exploitation of GTM. These, and issues still open, are discussed at the end of this work.

# Acknowledgements

This work would not have been possible without the support of many people. First of all, I am grateful to Prof. Dr. Theo Ungerer for letting me be part of research group and giving me the opportunity to conduct this work. His friendly manner and mentorship gave me good guidance during the course of this work. I am also grateful to Prof. Dr. Uwe Brinkschulte and Prof. Dr.-Ing. Rudi Knorr for being mentors for this habilitation project and their encouraging comments. I want to thank my former and current colleagues for good cooperations and fruitful discussion, and also for the good times we spent outside work. Some foundations for this work were laid during my stay in Toulouse in autumn 2012. I am greatly indebted to Prof. Dr. Theo Ungerer for making this stay possible, and to Prof. Dr. Christine Rochange, Prof. Dr. Pascal Sainrat, and Benoît Triquet for their friendly reception and the fruitful discussions we had (not only) during this time. I am also thankful to the students who contributed to this work through their bachelor or master theses, or their projects.

I am most grateful to my family, especially my parents Hildegard and Erich, for enabling me go to path, and their ongoing support and encouragement. Last but not least, I want to thank my friends for being constant companions and for the good times we have been spending.

# Contents

# List of Figures

# List of Tables

# List of Algorithms

# Part I.

# Baseline

# 1 Introduction

## 1.1. Motivation

Designers of safety-critical systems (SCSs) are building more and more powerful software. For example, in the automotive domain, sophisticated engine management systems allow to reduce fuel consumption and thus also reduce emissions. Advanced driver assistance systems increase the travelling comfort of a driver, for example through speed control or parking sensors. They also improve the safety of a car: Adaptive cruise control systems automatically keep a safety distance to the car running in front and thus help to avoid collisions (see e.g. Vahidi and Eskandarian 2003; Kesting et al. 2007). Pedestrian detection can help to prevent serious accidents between cars and pedestrians (Chiang et al. 2015). Furthermore, there are attempts to combine functionalities that are hitherto deployed to different dedicated electronic control units (ECUs) into a single ECU (Obermaisser et al. 2009). Today's cars often comprise 80 and more ECUs (Gut and Allmann 2012). Combining functionalities that are hitherto deployed to different ECUs onto a single *domain computer* will reduce the energy consumption, as fewer ECUs must be powered and the weight of the car is reduced.

Similar trends can be found in the avionic domain. There, weight reduction is an even more important driver for innovation. The foundations for a way to achieve this goal have been laid in 1990s through the introduction of the integrated modular avionics (IMA) concept (Prisaznuk 1992) that replaced the hitherto used federated architecture (FA) approach. In an FA, a large number of possibly small computers is distributed over an aircraft, each computer fulfilling only few dedicated control tasks. In IMA, applications are deployed to fewer, but more powerful computers, thus again reducing weight and power consumption. Like in the automotive domain, there is also a need for increasing performance in avionic computers, e.g. for new applications.

Due to the safety-critical nature of the whole system, it is necessary to ensure that applications integrated on a single computer cannot affect each other unpredictably. This is ensured through techniques for spatial and temporal *partitioning* (Rushby 1999), where each application is assigned a separate partition that is provided by a partitioning operating system (OS). Advancements and new requirements extend this concept to virtualisation (Popek and Goldberg 1974; Heiser 2008; Crespo et al. 2010): Partitions are provided by a hypervisor/virtual machine monitor (VMM). Inside each partition, either an OS is deployed on top of which applications are executed, or applications are run in a kind of bare-metal mode.

The trends to more powerful software inevitably require more powerful computer. This requirement is halfway met by processor technology: Moore's Law (Moore 1965) is still valid concerning the integration level of components on a single chip, meaning that the number of components on a chip still doubles each 12 to 24 months. However, the proportional advancements of software performance incurred a setback in the early 2000s. Until then, the sequential execution of programs was sped up by higher clock frequencies of new processor generations. At some point, further increases of clock frequencies were no longer feasible due to high energy consumption and power dissipation: processor development had hit the power wall (Asanovic et al. 2006). Chip manufacturers went over to exploit the increasing integration levels by deploying multiple processor cores on single chips. Thus, Moore's Law concerning processor performance was saved, but software development had to be, and is still being rethought to benefit from these performance increments (Sutter 2005): Software running on multicore processors has to efficiently exploit the parallelism of such systems, as no more gains of sequential execution are to be expected. However, basic performance gains for sequential execution could be achieved by assigning different applications to different cores of a multicore processor.

The developments of the past years have shown that the domains of general-purpose computing and non-critical embedded systems can cope well with this paradigm change. Meanwhile, developers of SCSs were in two minds about the use of multicore processors: On the one hand, such processors actually provide the performance required for the implementation of new safety features and more energy-efficient systems; on the other hand, multicore processors bring up a number of new problems for the development of SCSs (Kinnan 2009). Many of these stem from the nature of multicore processors where cores have to share at least some resource. Take, for example, two applications that hitherto were deployed to independent computers. Integrating these applications on two cores of a single multicore chip can result in both applications sharing a common memory bus. The memory access behaviour of one application thus can influence the behaviour of the other application. Such possible interferences must be analysed thoroughly to ensure that both application still can perform their duties correctly. In the meantime, efforts have been made to overcome these problems by appropriate hardware (Wilhelm et al. 2009; Cullmann et al. 2010; Ungerer et al. 2010; Bui et al. 2011) and software design (D'Ausbourg et al. 2011; Boniol et al. 2012; Nowotsch and Paulitsch 2012). First multicore processors are commercially available that explicitly target the domain of SCSs, e.g. the TMS570LS series by Texas Instruments (Texas Instruments 2011), or the AURIX family from Infineon (Infineon 2014).

This work goes one step further. Nowadays, advancements in processor technology allow to integrate tens and hundreds of cores on a single chip. Commercial prototypes like the Intel Polaris (Vangal et al. 2007) or the Intel Single-Chip Cloud Computer (Howard et al. 2010) demonstrated the feasibility of such approaches. In such *manycore* processors, the single cores no longer use a common bus, but instead are connected by a *network-on-chip (NoC)* over which they exchange messages. Such architectures are already being exploited commercially, e.g. through the TileGx processors with 9 to 72 cores by Tilera (acquired by EZchip in 2014) for network computing (Tilera 2011), through the Xeon Phi by Intel with over 50 cores for server computing (Intel 2014). Manycore architectures are also being established for embedded and real-time computing, e.g. through the MPPA-256 by Kalray (Dinechin et al. 2013) with 256 cores or the Epiphany architecture by Adapteva with 16 or 64 core (Adapteva 2013a; Adapteva 2013b).

The use of such processors in SCSs is to be expected in the future. To make their application feasible, a number of challenges must be overcome that arise from the special requirements of safety-critical domains. Thereby, the analysability of a manycore-based computer is the core challenge. In this context, the OS plays a key role as it represents the interface between application software and the underlying manycore hardware. Downwards, it has to manage the resources of the processor in an efficient and safe manner. Topwards, it provides the basic abstractions that software developers use to implement safety-critical applications. Both aspects, the hardware management and provision of abstractions, are tightly interwoven. The abstractions provide means for the application to use the underlying hardware. The OS has to ensure that any use of an abstraction (and the underlying hardware) by an application does not interfere with guarantees given to other applications. Furthermore, the abstractions must be designed such that they enable the development of efficient and safe applications.

## 1.2. Aims

The overall aim of this work is to leverage the use of manycore processors in future SCS by providing apt OS support. A general OS architecture provides the necessary baseline. Although not being in the center of this work, the aspect of virtualisation is also addressed. In the context of this OS architecture, relevant OS mechanisms are developed and examined for their use inside the SCS context. A guiding challenge throughout this work is the timing analysis of safety-critical software that must be supported efficiently. Necessarily, the OS itself must be timing analysable over its whole life cycle. While regular execution is being tackled in many research works, this work will consider border aspects of the life cycle, namely the boot process. On application level, the topic of timing analysis is closely interwoven with the question how resource sharing between applications is handled. This concerns, for example, the access to I/O ports that are shared between several applications, but also the sharing of single processor cores in the context of a multitasking system. Existing works on hard real-time scheduling provide good answers to these problems (C. L. Liu and Layland 1973; Sha et al. 2004), also in the context of multiprocessor systems (Davis and Burns 2011). However, such schedulers usually abstract from the concrete behaviour of applications resp. require that the application is pressed into the corset that is given by the scheduler's parameter set. If more knowledge about the application's timing behaviour can be obtained during its design process, a more flexible scheduling approach might be taken. Therefore, means for the modeling of timing behaviour of complex systems are considered. The use of information from such model is examined for a special case of real-time tasks: Even in the context of hard real-time systems, constraints can be more relaxed (see e.g. Jensen et al. 1985; Bernat et al. 2001). Therefore, scheduling techniques for tasks with relaxed real-time constraints are investigated.

## 1.3. Overview

The first part of this work lays the foundations for the investigation of manycore OS techniques. In the following chapter 2, the term "safety-critical system" is explained in more detail, thereby especially heeding the context of computers and OSs. It is followed by a discussion of manycore processors in the context of SCSs in chapter 3. The state of the art in manycore OSs is discussed in chapter 4. The second part of this work covers advancements

to the state of the art. Based on the outcomes of the discussions of the first part, a general OS architecture of SCS, MOSSCA, is introduced in chapter 5. Chapter 6 investigates the coordination and real-time capability of the boot process in manycore processors. A concept to capture timing properties of applications is presented in chapter 7. Its aim is to extract more knowledge from the design about timing behaviour of applications, which later can be exploited to optimise a system configuration. Chapter 8 studies the core-local scheduling of multitasking applications under relaxed real-time constraints. The third part concludes this work with an outlook on future work (chapter 9) and draws conclusions in chapter 10.

# 2

# Safety-Critical Systems

A system is classified as being *safety-critical*, if its malfunction can result in heavy or even catastrophic consequences. Among these are, obviously, bad injuries or deaths of humans, but also damages to the environment or to expensive machines. Computers controlling SCSs are ubiquitous. They can be found, e.g. in cars, air- and spacecraft, chemical and nuclear plants, or medical machines, thus controlling possibly critical aspects of everyday's life. However, engineers also have to consider that the computer may fail. A prominent and early example is the Therac-25, a computerised radiation therapy machine. Malfunction of this machine lead to overdoses of radiation being applied in at least six cases, resulting in deaths and serious injuries (N. Leveson and Turner 1993). Poor software engineering led to a failure of the Ariane 5 flight 501. This resulted in self-destruction of the rocket 37 seconds after its launch and a material loss of around 1.9 Billion French Francs (Lions 1996; Le Lann 1997).

This chapter addresses the term *safety-critical system* in more detail. The following section 2.1 presents generally used definitions and standards and gives a coarse overview, how these are implemented in practice. In section 2.2, the requirements on computers and especially software in SCS is discussed. This leads to the definition of OS requirements which are presented in section 2.3.

## 2.1. Definition and Realisation

Excluding mishaps completely is nearly impossible due to the complexity of SCSs and possibly unknown or unexpected operating conditions. Nevertheless, engineers strive to make such systems at least as safe as possible. This is reflected by the aim to keep the probability of mishaps below a certain bound. Depending on the severity of mishap consequences, this bound typically ranges from $10^{-5}$ to $10^{-9}$ over a given time span (N. G. Leveson 1986). For example, the IEC-61508 standard defines four *safety integrity levels (SIL)* with different mishap probabilities (Bell 2006). For low demand operations, the acceptable mishap probabilities range from $\left[10^{-2}, 10^{-1}\right)$ for SIL 1 down to $\left[10^{-5}, 10^{-4}\right)$ for SIL 4 per demand. For high demand or continuous operation, mishap probabilities from $\left[10^{-6}, 10^{-5}\right)$ for SIL 1 down to $\left[10^{-9}, 10^{-8}\right)$ for SIL 4 per hour are defined. The automotive safety standard ISO-26262 takes a slightly different approach (Hillenbrand 2012): *Hazards* that can cause a violation of the system's safety property, i.e. result in possibly catastrophic consequences, are classified according to the three criteria *severity of damage*, *probability of exposure*, and *controllability*

*by driver*. For each criterion, four to five classes are defined. Possible combinations of these classes are in turn classified into either one of the *automotive safety integrity levels (ASIL)* A to D, or *quality management (QM)*. Hazards classified in ASIL (with ASIL D being the highest level) require special measures to be taken to ensure the safety of the car.

Bounding the probabilities for mishaps requires two central steps to be taken (Dunn 2003): In the first step, possible hazards must be analysed. A failure mode analysis discovers all possible failure sources in the system that can lead to a certain hazard. The second step aims to mitigate the risk of mishaps. According to Dunn (Dunn 2003), this can be achieved in three stages:

- *Improving the reliability and quality* of components reduces the probability for component failure, thus reducing the risk of mishaps. Typically used approaches for reliability improvement are the employment of redundant components and the redesign of components. Systematic failures can be avoided by employing quality-oriented development approaches.
- To further reduce mishap risks, *internal safety devices* can be integrated into the system. They can reduce the effects of hardware and software faults, and systematic failures.
- To further counteract systematic failures, *external safety devices* can be employed, e.g. by physically containing the SCS.

At the end of this process stand proofs and documentation that the probability of failure is within the bound specified in the relevant safety standards. In highly critical domains like avionics, these documents are examined by an independent certification authority to ensure that all relevant safety standards are kept. In the automotive domain, a similar approach is taken today. However, instead of using an external certification authority, a company-internal *qualification* process is performed to ensure adherence to standards.

## 2.2. Computers and Software in Safety-Critical Systems

Computers in SCSs are embedded into a physical context, which is reflected by the term *embedded (computer) system*. Generally, this term embraces any computer that is embedded within a larger context, as opposed to, e.g. general purpose or high-performance computers. In the following, the term will be used in a restricted meaning to denote a computer that is embedded in the context of a safety-critical system. The job of such computers is to control physical processes and to react upon events originating in the physical domain.

A central requirement for such a computer follows from its safety property. It must provide a predictable behaviour such that it can be analysed in a feasible manner and within its physical context. Requiring a predictable behaviour of the deployed software is a direct consequence. This requirement can be decomposed into several aspects:

- Like from any other software, a *functional correct behaviour* is demanded. To meet the strict requirements of SCSs, axiomatic proof systems (Hoare 1969) and formal methods approaches (Clarke and Wing 1996) can be employed. These methods are outside the scope of this work. Additionally, a number of non-functions requirements apply:
- First among these is the need for a *correct timing behaviour*. The interaction with physical processes requires that reactions to events happen within a certain time span. For

example, the airbag in a car must be inflated within a few milliseconds after a collision is detected, else it would be worthless and the consequences for car occupants might be catastrophic. Thus, it is necessary that the execution time of software components can be upper bounded to ensure that deadlines are not missed. The usual approach is to perform a *worst-case execution time (WCET)* analysis (Wilhelm et al. 2008).

- An improved reliability can be achieved by equipping the system with *fault-tolerance properties* to mitigate the consequences of certain hazards. For example, to counteract failures of single computers in a distributed systems, computations can be executed redundantly on multiple nodes and results are compared by a voter.

Today, the analysis of embedded software is aggravated by the fact that an embedded computer usually executes not only a single application. Instead, multiple applications are deployed to a single computer and executed concurrently. In the avionics domain, this step was taken by the transition from the *federated architecture (FA)* to the *integrated modular avionics (IMA)* approach (Prisaznuk 1992). Similar efforts are undertaken in the automotive domain to replace multiple small *ECUs* by larger ECU *domain computers* (Obermaisser et al. 2009). Having multiple, independent applications running on the same computer, special heed must be payed to possible interferences arising from the sharing of resources. First, this can complicate or even render WCET analysis infeasible, if interferences cannot be predicted and bounded. Second, if interferences can occur in an unpredictable manner, the safety property of the system can be violated. Different stages of countermeasures are applied. Sufficiency of execution time can be guaranteed through a *schedulability analysis*. A combination with *watchdog mechanisms* can ensure that single applications cannot impede the execution of other applications through excessive resource reservations. *Memory protection* mechanisms prohibit applications from manipulating memory areas of other applications.

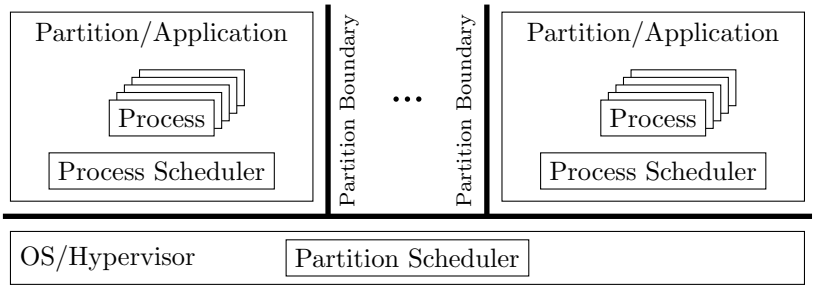



Figure 2.1.: Basic Application Architecture for one computer; Partition boundaries in thick lines ▬

*Partitioning* concepts (Rushby 1999) implemented in operating systems or hypervisors provide the strongest isolation of applications from each other. An exemplary software architecture for this concept is depicted in figure 2.1. The partitioning OS or the hypervisor resides at the bottom, running directly on the hardware. It provides software partitions which are isolated against each other in space and time. Space isolation is mainly ensured through memory protection mechanisms. The partition scheduler provides time isolation, as it has full control about all execution resources. The partition boundaries prevent unpredictable interferences and error propagation between partitions. Communication between partitions is performed over OS-provided primitives that decouple the involved partitions. Inside a par-

tition, either an application is executed directly, or a virtualised OS is deployed that manages several applications. The partitioning approach allows to develop and certify applications independently from each other.

Beyond the increased security for the single partitions (if the hypervisor is secure), using OS virtualisation has a number of further advantages for SCSs (Heiser 2008; Heiser 2011): It allows the integration of applications with heterogeneous requirements on a single computer. For example, time time-critical software using an real-time operating system (RTOS) can be combined with graphical user interface (GUI) applications running on top of a general-purpose OS. Used on manycore processors, the hypervisor can provide multicore partitions for the execution of software that cannot scale to the manycore level.

## 2.3. Operating System Requirements

OSs are influenced in two ways by the requirements of SCSs. As the OS itself is part of the SCS, it has to fulfil the requirement for a predictable behaviour itself. Additionally, it provides the foundations for the implementation of safety-critical applications. These foundations must be shaped such that adhering to the SC requirements in application implementation is eased or even enforced. Therefore, the following requirements on an OS for SCS can be postulated (Kluge, Triquet, et al. 2012):

**OSR-1 (Predictability and analysability)** The whole system must behave *predictably* and must therefore be *analysable*. This includes a *predictable timing behaviour* to ensure that all deadlines are kept.

**OSR-2 (Partitioning)** *Partitioning* in time and space guarantees freedom from interference. Special care must be taken to make accesses to *shared resources* predictable.

**OSR-3 (Communication)** Fine-grained communication takes only place between processes of the same application. If applications have to exchange data over partition boundaries, special mechanisms are provided by the OS.

**OSR-4 (Reconfiguration)** There is ongoing research on the dynamic reconfiguration of software in safety critical systems (Pagetti et al. 2012). While this is not part of today's standards, the capability for *reconfiguration* will be an important requirement for future SCSs.

These requirements form the side conditions of the work at hand. Techniques presented in later chapters must adhere to them.

# 3 Manycore Processors

Over the last decades, advancements in processor manufacturing allowed to integrate more and more transistors on single chips. Until the early 2000s, these increases of integration density were mainly used to increase the performance single core processors. After having hit a *power wall* (Asanovic et al. 2006; Asanovic et al. 2009), further increases of clock frequency were no longer possible due to high power dissipation. Just then, processor producers passed on to exploit the increasing integration density by integrating multiple processor cores on single chips. Today, hundreds of cores can be integrated in single chips. In the future, developers of SCSs will have to use such manycore processors to meet the performance demands of their applications. On the one hand, this poses a challenge, as software models and development processes must be rethought. On the other hand, development of SCS computers can benefit from manycore processors, as the spatial separation of the cores can provide a sound base for the partitioning concepts used in these domains. Insofar, it is important to examine the properties of manycore processors and their impacts on OSs for SCSs.

Section 3.1 gives a coarse overview of the current state of manycore processor technology. In section 3.2, the properties of manycore processors are discussed and set into general context of computer systems. Section 3.3 provides a view on manycore processors from the special viewpoint of SCS. Additional requirements for OSs for manycore processors are discussed in section 3.4.

## 3.1. State of the Art

In early multicore processors, the single cores are connected by a shared bus that also provides access to off-chip devices like the shared memory. With increasing core numbers, the bus concept does not scale well. Multicore processors with moderate core numbers can use crossbars instead (Kongetira et al. 2005). With increasing core numbers, the crossbar concept scales poorly in terms of space. For large numbers of cores, NoC-based approaches are used (Hemani et al. 2000; Dally and Towles 2001). Today, such NoCs structures can be found in several commercial processors. In the EZChip TileGx processor family, up to 72 cores are connected by five separate NoCs (Tilera 2011). In a similar manner, the Epiphany cores of the Adapteva Parallella are connected by three separate NoCs (Adapteva 2013c). Intel uses a ring-based interconnect on its Xeon Phi (Intel 2014).

Also, processor manufacturers are using hybrid interconnection architectures. Such an approach can be found e.g. in the MPPA-256 processor from Kalray (Dinechin et al. 2013): The 256 cores of the processor are grouped into 16 *compute clusters*. In each cluster, 16 cores are connected to a shared memory. Additionally, the cores in each cluster share NoC routers that enable communication with other clusters.

Currently, only few multicore processors exist that explicitly target SCSs. The Hercules TMS570 family by Texas Instruments integrates two ARM Cortex-R cores that can run in lock-step mode (Texas Instruments 2014). These microcontrollers are developed specifically under the influence of current safety standards and meet the requirements of ISO 26262 ASIL-D and IEC 61508 SIL-3. The same standards are also targeted by Infineon with their the AURIX microcontrollers (Infineon 2014). The AURIX architecture contains three TriCore cores, two of which can be run in lockstep mode. Furthermore, there are efforts to also use commercial off the shelve (COTS) multicore processors in safety-critical systems (Pellizzoni et al. 2011; Boniol et al. 2012; Nowotsch and Paulitsch 2012; X. Wang et al. 2012). For space missions, an adaptation of the Tilera Tile64 processor has been discussed (C. Y. Villalpando et al. 2010; C. Villalpando et al. 2011).

Several research projects have investigated the development and use of multi-/manycore processors for safety-critical real-time systems. In the MERASA project, a real-time capable bus-based multicore processor with up to 8 simultaneous multithreaded cores has been developed (Ungerer et al. 2010). In the ACROSS project, a multicore processor was designed specifically for SCSs (Salloum et al. 2012). An important objective of this work was the temporal determinism of the platform. The T-CREST project (T-CREST 2013) has developed a time-predictable NoC-based multicore processor and appropriate compiler and worst-case execution time (WCET) analysis support. In the scope of the project parMERASA (Ungerer et al. 2013), a multicore processor with distributed shared memories was developed and parallelised industrial real-time applications were deployed successfully to this processor. Recently, Kalray has announced their Bostan (MPPA2-256-N) manycore processor. Bostan is designed such that it can deliver a predictable timing behaviour on the level of cores, compute clusters, and the NoC (Dinechin 2015). Further works on an OS for the Bostan aim to provide a manycore platform that enables the certification of application running on it (Saidi et al. 2015).

## 3.2. Architecture Characteristics

In their general structure, NoC-based manycore processors resemble the concept of a distributed system: They consist of multiple processing units that can communicate over some kind of interconnection network. Nevertheless, there are big differences in important details: The single nodes of a classic distributed system are usually powerful computers with large memories. In contrast, the cores of manycore processor possess only small local memories with short response times, while accesses to big (off-chip) memories are rather expensive. Communication between nodes on a manycore chip is rather cheap compared to distributed systems, where large network stacks and comparably slow networks must be traversed. In general, the following key properties can be attributed to a manycore architecture:

**Fast communication** Small messages between cores are transmitted with low latencies. Typical traversal times are in the range of few cycles per hop from a router to the next one.

**Small local memories** Due to limited chip area, each core possesses only small local memories that can be accessed fast. These can either be local caches like in the EZChips Gx Family (Tilera 2011), or addressable memories like in the Adapteva Epiphany (Adapteva 2013c).

**Expensive large memories** Large memories are available, but cannot be integrated on the same chip. Therefore, access to such memories is rather expensive in terms of access time.

Obviously, to exploit the performance of manycore processors, software development must undergo a paradigm shift. Most benefits is gained for algorithms that can be split into small parts of code that fit into the local memories. Similar restrictions apply for the data that the code operates on: Each core-local computation should require only small portions of data that fit into the local memory. 3D stacking techniques may alleviate this restriction in the future (see e.g. Black et al. 2006; Loh 2008; Lim 2013). Benefit can be drawn from the fast communication, as it allows a fast exchange of results with other computations. However, use of a global off-chip memory should be minimised due to the large access penalties. In summary, the greatest benefit can be drawn from a manycore processor, if the programs that are executed on the single cores are as specialised as possible.

The lack of I/O capabilities that are typical for real-time embedded system (RTES) is yet impeding the use of manycore processors in this domain. Most of today's manycore processors are not targeted for RTES, but rather for network computing (like the EZChips Gx Family) or as accelerators for general-purpose computing. Often, interfaces for external memories, ethernet, or PCIe can be found. Interfaces used in RTES, like UART or SPI, can so far only be found in the MPPA-256 by Kalray.

## 3.3. Manycore Processors and Safety-Critical Systems

While widely used in the domain of general-purpose computing, multicore processors are only slowly entering the domain of SCSs. In 2009, Kinnan (Kinnan 2009) identified several issues that are preventing a wide use of multicore processors in safety-critical RTESs. Most of these issues relate to the certification of shared resources like caches, peripherals or memory controllers in a multicore processor, but also to the power feeds and clock sources. Wilhelm et al. (Wilhelm et al. 2009) show how to circumvent the certification issues of shared resources through a diligent hardware design. Later work also shows that even if an architecture is very complex, smart configuration can still allow a feasible timing analysis (Cullmann et al. 2010). Additionally, Kinnan (Kinnan 2009) also identified issues that naturally inhere the concept of multicore. The replacement of multiple processors by one multicore processor can introduce the possibility of a single point of failure for the whole system: Separate processors have separate power feeds and clock sources, where the failure of one feed will not impact the other processors.

The problems discussed above stem mostly from the fine-grained sharing of many hardware resources in today's multicore processors. On the single-core processors used in today's SCSs, the partitions executed on one computer share one core. Even with multicore computers, several partitions would have to share one core and various common resources. With an increasing number of cores, it would be possible to assign each partition its own core or even a set of cores exclusively. Future SCSs can greatly benefit from manycore architectures, as resource sharing would be reduced. The only shared resources are the NoC interconnect

and off-chip I/O. The aim of this work is to develop a system and OS architecture that can provides safety-criticality isolation properties on such processor and make the processor usable for future safety-critical applications.

Concerning power feeds or clock sources, manycore processors also possess single points of failure (Kinnan 2009). These problem cannot be resolved by software design. However, if hardware faults occur on the chip that only impact a delimited part of the processor, it may be easier on a manycore processor to cope with such a fault. The single nodes in a manycore processor are coupled less tightly through the NoC than in a bus-based multicore. Thus, fault-tolerance mechanisms will gain a higher potential for a reconfiguration of the computer to keep up operation of critical applications.

## 3.4. Operating System Requirements

Beyond the general requirements of SCS (see sect. 2.3), an operating system for SCS based on manycore processors must also accommodate for the special properties of manycore processors.

**OSR-5 (Code size)** OS code that is executed on any node must be as small as possible. In systems with local addressable memories, more space is left available for application code. If the single cores only have caches, cache poisoning through OS calls is diminished, thus easing the timing analysis of application code.

**OSR-6 (Shared data)** Sharing of large data structures between OS modules executed on different nodes should be avoided. Thus, possible interferences between different nodes are attenuated.

These requirements extend the side conditions for this work that were defined in section 2.3.

# 4 State of the Art in Manycore Operating Systems

Research in manycore OSs mostly tackles the general-purpose and high performance domains. So far, the special requirements of SCSs are not seen as a central theme. The following section 4.1 gives an overview of current research approaches on manycore operating systems and opposes these approaches to the OS requirements from sections 2.3 and 3.4. Section 4.3 provides conclusions on the state of the art.

## 4.1. Existing Approaches

### 4.1.1. Corey

The Corey operating system targets cache-coherent shared memory multiprocessors. Boyd-Wickizer et al. (Boyd-Wickizer et al. 2008) have shown that applications on such processors can suffer notable performance losses if kernel data structures are shared over several cores. Further problems arise from so-called *TLB shootdowns* where the translation look-aside buffers (TLBs) of all cores are flushed. Corey aims to solve these problems by giving applications control about sharing of operating system data structures. Basically, data structures should be used on only one core. If sharing is necessary, the OS provides means allowing the initiator to control the sharing.

#### Design Principles

The Corey OS is based on the exokernel principles (Engler et al. 1995), where physical resources are managed on application level. Applications can control sharing of OS data through the following three abstractions:

**Address ranges** are used to implement a distributed shared memory where applications can exchange data. Applications can selectively share parts of their address space with other applications, while keeping other parts private. Such sharing of the address space helps to avoid poisoning of caches and TLBs, as hardware page tables will also be shared.

**Kernel cores** exclusively execute kernel code. Thus locality of kernel code and data is achieved. Furthermore, the kernel can relinquish locks, and replication of kernel data

and thus cache misses are avoided. Kernel cores are used for operating system management and I/O. Applications running on other cores can call kernel services via shared memory inter process communication (IPC).

**Shares** allow to define more clearly, how far kernel lookup tables are shared. A *share* is a mapping of identifiers visible by applications onto specific kernel data pointers. Manipulation of a share would require locks. These should occur at the least global scope the data is used. Shares are built in a hierarchical way, and the root share of an application is private. Applications can create new shares under their root share, which they possibly share with other applications.

#### Structure and Implementation

The above abstractions are integrated in the Corey kernel through special objects, namely *shares*, *segments*, *address ranges*, *pcores* and *devices*. The Corey kernel provides a set of system calls that allow to manipulate these objects. The kernel locks only the metadata of objects it actually accesses, thus relieving itself from lock contention. *Shares* are used by applications to control and manage access to any kernel object. Memory management is performed by mapping physical memory *segments* to *address ranges*. A *pcore* object provides an abstraction of a physical core allowing management at application level. Applications can add virtual *devices* to their pcore. They act as an interface to the actual physical device that is managed by a kernel core. Applications invoke system calls at a kernel core through a special *system call device*. The kernel provides an automatic garbage collection on objects that is based on reference counting. Based on these low-level functionalities, library operating systems can provide high-level OS services.

Corey itself exports three basic system services. First, application can call a `cfork` service similar to the Unix `fork` to extend their execution an a new core. The calling application can pass own shares or address ranges to the new processor. Second, the lwIP library (Dunkels et al. n.d.) makes it possible to run several network stacks and to even virtualise the network card. Finally, an inter-core shared buffer cache provides a communication facility. It incorporates a lock-free tree, a write scheme to minimise contention, and a scalable read-write-lock.

Corey was implemented for AMD Opteron and Intel Xeon multicore processors. Use cases for the evaluation were a MapReduce and a webserver application.

#### Discussion

The work on Corey shows the possibly detrimental performance effects of sharing data structures between cores, even if cache-coherence mechanisms are available. Corey solves these problems by controlling the sharing of data through the abstractions of shares and address ranges, which are similar to the concept of *address ranges* from the µKernel paradigm (Liedtke 1995). Insofar, this work emphasises OSR-6 (Shared data). The requirements of SCS are not addressed by Corey.

### 4.1.2. Barrelfish

Schüpbach et al. (Schüpbach et al. 2008) identified three types of diversity that distinguish multicore processors from singlecore systems: (1) Memory architectures trend towards *non-uniformity* to allow scalable multiprocessing. (2) Heterogeneous multicores introduce *core*

*diversity*. (3) Different systems have different resources available *(system diversity)*, thus making it hard to define general code optimisations. The Barrelfish OS embraces this diversity with its Multikernel concept (Baumann, Barham, et al. 2009). It is guided by the idea that today's computers already are distributed systems, and thus the OS should be reconsidered to exploit this fact for future hardware (Baumann, Peter, et al. 2009).

### Design Principles

Barrelfish is based on the following concepts (Baumann, Barham, et al. 2009):

**Explicit communication** between cores avoids the problems coming with shared memory, like contention of caches and the coherency fabric. It allows to also integrate cores into a system that are not cache-coherent or even have no shared memory. If shared memory is available, it is only used to implement message channels. This message passing approach allows to map the distributed nature of a system to the application layer. The authors argue that using messages comes at a lower cost than shared memory and scales better with increasing core numbers. Also, this modularised approach is amenable for analysis as a theoretical foundation therefore already exists, e.g. Hoare's communicating sequential processes (Hoare 1978) or $\pi$-calculus (Milner et al. 1992). Furthermore, such a system can be refined more easily, and it can also be built fault-robust more easily.

**A hardware-neutral** OS structure aims to separate hardware related mechanisms from OS mechanisms. The hardware mechanisms, like message transport and interfaces for CPUs and devices, are implemented in a small kernel that provides an abstraction layer. This small interface allows a fast adaption to new hardware platforms. All OS mechanisms are implemented uniformly on top of this kernel.

**Replication** of state instead of sharing is an automatic consequence of the message passing architecture of Barrelfish. If some local instance of the OS has to access data that is possibly shared, this data is treated as a local replica. Consistency between the OS instances is ensured through messages. This approach shall improve system scalability and reduce the load on the interconnect, memory contention and synchronisation overhead. Through state replication it is also easier to support changes in the set of running cores, e.g. when switching off parts of the system to reduce energy consumption, or when hotplugging processors. On shared-memory processors, it is also possible to privately share a replica of the system state within a group of processors or cores.

Peter et al. (Peter et al. 2010) point out further requirements that Barrelfish fulfils to effectively embrace diversity in hard- and software. They show that time-multiplexing of resources will still be necessary in future systems to heed the differing requirements of application software. However, scheduling should be performed at multiple timescales, namely long-term application placement, medium-term resource-allocation, and short-term thread scheduling. To ease portability of the OS, and for efficient application execution, the OS must further be able to reason online about the underlying hardware and the application as well. Furthermore, OS and applications must able to communicate in a two-way manner, e.g. for allocation of resources or improvement of scheduling efficiency.

#### Structure and Implementation

Barrelfish is a Multikernel operating system (Baumann, Barham, et al. 2009). Architecture-specific *CPU drivers* provide basic abstractions from the underlying hardware. They are similar to the exokernel concept (Engler et al. 1995). All higher-level OS mechanisms are implemented as user-level processes. *Monitors* implement functionalities for a system-wide state, message passing, and mechanisms and policies of traditional single-core kernels.

Barrelfish's process structure differs notably from the Unix model. The CPU driver schedules so-called *dispatcher objects*, which themselves run core-local user-level thread schedulers. Therefore, Barrelfish employs a mechanism based on scheduler activations (Anderson et al. 1992).

For inter-core communication, Barrelfish implements a user-level remote procedure call (RPC) mechanism (Bershad et al. 1991) which is optimised for distributed caches. Memory management is performed by user-level code. The CPU driver only checks the decisions with the help of a capability system. To support concurrent application programs, Barrelfish also implements a virtual address space spanning multiple cores.

Barrelfish employs a *system-knowledge base (SKB)* to gather and manage information about the underlying hardware. In a similar way, applications present a *scheduling manifest* to Barrelfish containing information about the application's resource requirements. Barrelfish uses the information from SKB and scheduling manifest to optimise the resource allocation for applications.

Barrelfish was implemented for several Intel and AMD multicore systems. In 2011, an implementation for the Intel SCC was presented (Peter et al. 2011).

#### Discussion

The Barrelfish authors mention that by restricting communication to explicit messages they make the system and applications amenable for analysis (Baumann, Barham, et al. 2009). Insofar, Barrelfish can meet the SCSs requirements OSR-1 (Predictability and analysability) and OSR-3 (Communication) at least halfway at the functional level. The decoupling through messages can also provide a base for OSR-2 (Partitioning). Sharing of data is avoided, however at the cost of data replication. Furthermore, no information exists about the size of the OS code and its real-time capability. The implementation of OSR-4 (Reconfiguration) in Barrelfish is open.

### 4.1.3. Factored operating system

The *factored operating system (fos)* is motivated by today's cloud computing utilising thousands of cores as well as future manycore processors with possibly thousands of cores (Wentzlaff, Gruenwald III, et al. 2010). Wentzlaff et al. (Wentzlaff and Agarwal 2009) identified several points that hinder scaling today's operating systems to many cores. Among these are the use of hardware locks, aliasing in cache when OS and applications are executed on the same core, and the reliance on shared memory. fos aims to overcome these problems through (1) avoiding hardware locks, (2) the separation of execution resources and (3) avoiding a global cache-coherent shared address space. Therefor, fos draws inspiration from the concept of internet servers: Operating system services are factored into communicating servers that collectively provide a system service.

### Design Principles

fos is based on the following design principles (Wentzlaff, Gruenwald III, et al. 2010):

**Space multiplexing** The core principle of fos is to *replace time multiplexing by space multiplexing* of execution resources. In traditional computer systems, a processor core is shared by the OS and application processes. Each process gets a slice of the available processing time. fos draws benefit from increasing core numbers by allocating each OS or application component its own core exclusively. The computer system itself becomes a distributed system. Thus, execution resources of different components are spatially separated, and conflicts originating from resource sharing can no longer occur.

**Factoring of OS services** The OS itself is factored into function specific services, thus easing distribution of the OS on several cores. Communication and collaboration between services is restricted to passing messages. This approach makes sharing of data much more visible to the programmer.

**Adaption of resource utilisation** The OS monitors the utilisation of services and supplies more resources for services with a high load.

**Fault detection** fos utilises watchdog processes to monitor OS services. If a service failure is detected, fos starts a new service instance.

### Structure and Implementation

The fos design principles are implemented using the concepts of messaging, fleets, and naming (Wentzlaff, Gruenwald III, et al. 2010):

**Messaging** Inter-process communication in fos is strictly based on messaging. This does not require a shared memory, but can be implemented on top of it. It can also be provided by the underlying hardware. Also, messaging makes sharing of data more explicit to the programmer, thus requiring careful thought about the degree of shared data. fos still allows shared-memory multithreading to support legacy applications. Nevertheless, the implementation of fos itself is completely based on messaging. The restriction of fos to messaging enables an application developer to use a single communication API without having to care about the placement of processes in the actual system.

**Fleets** Services can be provided by fleets (see also (Wentzlaff et al. 2011)). They provide scalability for growing systems by grouping several servers of the same type. Fleets can adapt to changing load by growing (i.e. starting new servers) or shrinking, thus providing elasticity. A self-awareness property of fleets enables a fine-grained adaptation of a fleet's behaviour, e.g. by migrating servers or even single transaction.

**Naming** Services in fos are addressed through named mailboxes. These names are URIs similar to web addresses. They are managed by a dedicated nameserver. If multiple processes provide the same service as a server fleet, they register with the same name at the nameserver. Upon an application request for a specific service, the nameserver provides the fleet member best suited for this request. The metrics for this decision can be e.g. load balancing between servers or low communication latencies between application and server.

On each core, fos provides an identical µKernel. The µKernel is responsible for management of the local node. Additionally, it provides reliable messaging, named mailboxes and name mappings for communication. The name mappings are handled in cooperation with a fos server. A library layer (*libfos*) abstracts messaging and provides an interface for applications to communicate with system services. Applications can execute on one or more cores. fos provides a traditional system call interface to application writers. An RPC code generator translates these system calls into messages. To ease application development, fos also provides a library for distributed data structures.

fos servers work transaction-oriented and implement only stateless protocols. Transactions cannot be interrupted, thus removing the need for local locks in most cases. Long latency operations are handled with the help of so-called *continuation* (Draves et al. 1991) objects. While the operation is pending, the server thus can perform other work. The fos servers implement a cooperative multithreading model. Each server request generates a new thread that is executed until it explicitly yields when, e.g. waiting for replies from other servers. Similar to system calls, the interface to a server consists of function calls. A code generation tool generates code for serialising the parameters into messages and deserialising of the message. Thus, the message-passing nature of the system is mostly hidden from the application programmer.

A variety of high-performance message passing mechanisms have been developed (Belay 2011) for fos. These allow transparent messaging even within environments with a heterogeneous communication infrastructure like distributed multicore computers.

Several case studies demonstrate the behaviour of fos' key components. These are the implementation of a file system to demonstrate interaction of multiple servers, the mechanism for spawning servers in a computing cloud, and an elastic fleet to demonstrate the scalability and adaptability aspects of fos.

fos is implemented for x86_64 hardware as a para-virtualised machine operating system. Like Corey, it uses the lwIP library (Dunkels et al. n.d.) as network stack. Early evaluations were performed on a 255 core QEMU simulator, later versions of fos run on a cluster with Intel Xeon multicore processors.

### Discussion

The basic design principles of fos provide a sound base for SCSs. OSR-2 (Partitioning) is supported by space multiplexing applications, as each application can end up on a separate core. Similar to Barrelfish, OSR-1 (Predictability and analysability) and OSR-3 (Communication) are supported by restricting any communication to messages. The fulfilment of OSR-4 (Reconfiguration) is demonstrated by the concept of fleets. OSR-5 (Code size) and OSR-6 (Shared data) are also supported by the fos approach. Still, OSR-1 (Predictability and analysability) cannot be deemed as fulfiled completely, as the real-time capability of the used concepts is not considered. The provision of a functional interface for code development and tools that automatically convert these function calls to messages improves the usability of fos. However, the functional interface hides the message passing nature of fos, which slightly contravenes the aim of making the sharing of data more explicit through the exclusive use of messages.

### 4.1.4. Tessellation

Future manycore processors will not only be used in server systems, but also single-user client devices ranging from personal computers to handhelds. With their Tessellation operating system, Liu et al. (R. Liu et al. 2009) tackle the multifaceted requirements that prevail in this domain: the devices will simultaneously run parallelised applications with different characteristics, e.g. interactive or real-time applications. Many devices are powered by battery, thus making energy efficiency in OS power management an important factor. Tessellation is developed to meet a wide variety of performance goals.

#### Design Principles

Tessellation is based on the following principles:

**Space-time partitioning (STP)** is the core concept of Tessellation (R. Liu et al. 2009). A *spatial partition* as isolated unit is defined as a subset of physical resources. Space-time partitioning is achieved by time-multiplexing spatial partitions on the physical hardware on a coarse granularity. Liu et al. (R. Liu et al. 2009) argue that STP will be beneficial from several aspects. From the performance aspect, the scheduling needs of applications differ, and the parallelism within applications may require predictability. Spatial partitioning gives exclusive access to resources and thus can increase performance and predictability. Partitioning can also be beneficial from the energy aspect: if energy is scarce, partitions executing less critical applications can run with constricted performance or even be switched off, assuming appropriate hardware support. Partitions can support QoS guarantees, and help increasing correctness of application execution through resource isolation between applications. Finally, by splitting an application into *sub-application* partitions, STP can support the hybrid behaviour of applications, e.g. by executing untrusted browser plugins in a separate partition.

**Partition Interaction** is provided under the following aspects (R. Liu et al. 2009): (1) Inter-partition communication is based on passing messages over *inter-partition channels*; (2) Monitoring of channels and profiling of shared services ensure *cross-partition QoS*; (3) *Partition virtualisation* enables multiplexing of partition on the physical hardware and verification of all partitions' QoS requirements.

**Cells** are introduced by Colmenares et al. (J. A. Colmenares et al. 2010) as an abstraction for space-time partitions. They act as containers for parallel software components and provide additional guarantees for resource accesses.

**Two-level scheduling** of resources separates resource allocation to cells from resource utilisation within cells (J. A. Colmenares et al. 2010). The OS only decides which and how much resources to allocate for a cell. An application-specific scheduler takes care about the resource management within the cell.

#### Structure and Implementation

The Tessellation kernel (R. Liu et al. 2009) is based on several hardware partitioning mechanism. Some of these are already widely used today, like page tables and TLBs. Others are required for a safe partitioning of further shared resources, like caches or network bandwidth.

Tessellation would also profit from hardware support for message channels that can be used directly by unprivileged code. Furthermore, hardware should also provide mechanisms like performance counters to monitor the system.

The Tessellation kernel itself is split in two layers. The *Partition Mechanism Layer* forms the lower part of the kernel. Its implementation is machine-dependent and provides a machine-independent interface to the upper layer. The partition mechanism layer configures the partitioning mechanisms provided by the hardware. Other mechanisms are implemented through a form of paravirtualisation. The *Partition Manager* forms the upper layer of the Tessellation kernel. It is responsible for the scheduling of partitions. It schedules the partitions in space and time by allocating the partitions and time-multiplexing them on the physical hardware. The partition manager also allocates all resources and their amounts used by a partition using an abstract specification provided by the partition mechanism layer. Further tasks of the partition manager are the resizing of partitions, and balancing system performance with energy consumption. For fixed-sized partitions, the partition manager allocates the needed resources once at partition creation. For dynamic-sized partitions, an API must be implemented by the application that allows to dynamically add or remove resources from the partitions. Tessellation exports further interfaces for resource accesses, influence scheduling, and establish secure communication channels with other partitions. System services are provided by service partitions running an operating system like FreeBSD that can be accessed via messages.

The resource allocation architecture (J. A. Colmenares et al. 2010) in Tesselation is based on a *Space-Time Resource Graph (STRG)*. The STRG maps resources to the cells that are using these resources. A *Policy Service* implements admission control to decide about the admission of new cells and to maintain QoS guarantees. The policy service is also responsible for resource adaptation. It monitors the applications' behaviour and adapts their resource allocations such that they can meet their performance goals. To improve resource adaptations, the policy service maintains a model of the applications' behaviour. Starting with a generic model, this is refined by online training using the applications' actually monitored behaviour. In the *Partition Mapping and Multiplexing Layer*, an extension of the aforementioned partition manager, space-time scheduling of partitions is provided. A planner generates resource allocation plans based on an STRG that are implemented by a plan executor. The mapping layer realises several *cell activation policies* for e.g. binding a cell to specific cores, or time-periodic activation of cells.

Tessellation uses the Lithe framework for two-level scheduling of resources. Pan et al. (Pan et al. 2009; Pan et al. 2010) introduce the notion of a *hart (hardware thread)* as a low-level abstraction of real hardware with a 1:1-mapping between harts and physical execution resources. Harts allow a cooperative sharing of execution resources between libraries, and also space and time multiplexing by an operating system. Lithe provides an interface for sharing harts. Any library is seen as a scheduler for harts, and the library writer controls the hardware resources via the Lithe interface, resulting in a cooperative threading model.

Experimental evaluation of Tessellation was performed on a RAMP Gold simulator (64 SPARC V8 cores) and on a dual Intel Xeon quadcore platform using the PARSEC benchmarks (J. A. Colmenares et al. 2010) and a real-time musical applications (J. Colmenares et al. 2011). Further experiments were performed on an Intel i7 quadcore running a parallel soft real-time GUI service (Kim et al. 2012).

#### Descendants

The works on Tessellation and Lithe gave rise to two more operating systems. With the operating system ROS, Klues et al. (Klues et al. 2010) introduce the *ManyCore Process (MCP)* as central abstraction. Threads within a MCP are scheduled in userspace, thus removing the need for corresponding kernel threads and making the kernel more scalable. Physical resources used by a MCP must be explicitly granted and revoked. They are managed within so-called *resource partitions*. The resources are only provisioned, but not allocated. ROS guarantees to make them available if requested. While they are not used by the partition, ROS can grant them temporarily to other partitions. Like Tesselation, ROS targets general-purpose client computers.

The MCP concept is reused and extended by Rhoden et al. (Rhoden et al. 2011) in the Akaros operating system. System calls in Akaros are asynchronous to decouple I/O operations from parallelism. The Akaros kernel provides multiple scheduling granularities for cores. Cores with coarse-grained scheduling at time slices of about 100 ms provide good cache performance. Fine-grained scheduled cores with time slices $\leq 1$ ms are used for operations requiring a low response time. Furthermore, the Akaros kernel supports block-level data transfer with zero-copy I/O primitives between disks and/or network devices.

#### Discussion

The requirements OSR-1 (Predictability and analysability) and OSR-2 (Partitioning) are partially met by Tessellation due to its space-time partitioning. However, the QoS guarantees provided to the partitions may not meet the strict requirements of SCSs. The partition interaction over monitored channels can fulfil OSR-3 (Communication), if the QoS guarantees are given in an appropriately strict manner. The concepts of Tessellation should allow to implement mechanisms that fulfil OSR-4 (Reconfiguration). No information is available on the code size of Tessellation, and how kernel data is managed.

### 4.1.5. Helios

Although not explicitly targeting multi-/manycore computers, the Helios operating system introduces concepts that might be transferred to this domain. Nightingale et al. (Nightingale et al. 2009) target with their work heterogeneous multiprocessing platforms that consist of regular CPUs as well as additional programming devices, and that possibly only implement a non-uniform memory access (NUMA) model. Helios is based on four design goals that shall ease application development for these platforms: (1) Even though the underlying hardware is heterogeneous, the kernels running on the nodes shall provide a uniform set of abstractions. (2) Communication between processes shall be transparent, regardless whether they are running on the same or on different nodes. (3) Application deployment and tuning shall be simplified by relieving the developer partially from the burden of process placement. (4) Processes shall be executable on multiple platforms, but execution shall also be able to exploit platform-specific features.

#### Design Principles

The core concept of Helios are *satellite kernels*, that are a special kind of microkernels. They are designed such that they minimise costly remote communications. A satellite kernel re-

quires minimal hardware primitives and resources and thus can be executed on a wide range of programmable devices. Process-local resources are managed by the kernel via fast system calls. Shared resources are managed by a dedicated process that can be accessed through an IPC mechanism.

Transparent IPC in Helios is based on the concepts of *namespaces* and *local and remote message passing channels*. Namespaces are used to determine communication endpoints regardless on which physical resource these endpoints reside. The message passing channels are both accessible through one IPC abstraction, and the Helios satellite kernel decides which mechanism is actually used. For local message passing, a fast zero-copy mechanism is used. Remote message passing is implemented on top of mechanisms available in the concrete hardware, including memory and PCI buses.

For process placement, processes in Helios can specify an *affinity* for other processes. If a process requires fast communication with another process and thus should be executed on the same satellite kernel, this is represented by a positive affinity. Processes can also express platform preferences by positive affinities for specific satellite kernels. In contrast, a process expresses a negative affinity if it should be executed on a different satellite kernel to avoid e.g. resource contention. Affinity values are provided as metadata, and thus can easily be changed while tuning a system.

Helios achieves platform-independence through a two-phase compilation strategy. Applications are shipped in an intermediate language representation. In a second compilation phase this code is translated into a binary for the respective execution platform. The alternative of shipping fat binaries that contain code for several platforms is discarded for reasons of code size and flexibility.

#### Structure and Implementation

The implementation of Helios is based on the Singularity operating system (Hunt et al. 2005) and was performed by modifying the Singularity Research Development Kit (RDK) (Microsoft Corporation 2008). Helios builds on the fast message passing and context switch mechanisms provided by Singularity, thus promoting the a modular programming model where processes communicate via messages. Helios extends the homogeneous symmetric programming model of Singularity to heterogeneous platforms with programmable devices and NUMA domains. Each NUMA domain executes its own satellite kernel. A process can reside in only one NUMA domain. Hence, the memory accesses of a process can be guaranteed to be only local.

Helios processes are written in the Sing# programming language (Fähndrich et al. 2006) and are compiled into the common intermediate language (CIL) of the the .NET platform. The Bartok compiler, a derivative of Marmot (Fitzgerald et al. 1999), compiles the final binaries for a particular platform.

The boot process of Helios is coordinated by the kernel that starts first *(coordinator kernel)*. This kernel examines the system for other programmable devices and initiates the loading of further satellite kernels. The coordinator kernel is also responsible for the management of the namespace. Applications can register services in the namespace, thus making them available for other applications. The coordinator kernel associates such an entry with a dedicated message passing channel. Other processes can use this channel to bind to services by requesting a dedicated message passing channel with the associated application. Entries in the namespace are removed if explicitly requested by the advertising service or on closure of the associated message passing channel.

Each satellite kernel possesses an *exchange heap* that is dedicated to remote message passing. Outgoing messages are stored on this heap and message metadata is transmitted through a shared FIFO buffer. The receiving satellite kernel evaluates the metadata and copy-pulls the message into its own exchange heap.

Affinity of processes is specified in an XML manifest. The manifest contains platform, positive and negative affinity. Helios uses these values in the given order to determine the satellite kernel a process will be executed on.

So far, satellite kernels for Intel XScale and NUMA machines have been implemented. Evaluations were performed on Intel Core2 Duo with an Intel XScale processor attached, and on dual-processor system with AMD Opteron dual-cores. Evaluations included file systems and several networking applications like network drivers and web servers.

#### Discussion

Although Helios introduces concepts that can be useful for manycore processors in general, their application in SCSs is not recommended. Helios is designed to be used in flexible and dynamic systems, which in principle would support OSR-4 (Reconfiguration). The degree of flexibility provided by Helios cannot be found in SCSs, as it would render a thorough analysis of the system infeasible. Nevertheless, the concept of satellite kernel gives a hint on the possible fulfilment of OSR-5 (Code size).

### 4.1.6. Osprey

Sacha et al. (Sacha et al. 2012) devise the Osprey operating system. Their aim is a fault-tolerant operating system that allows a predictable execution of real-time applications in computing clouds on NUMA multicore computers. To achieve these goals, they use the concept of OS virtualisation where the virtual machine interface is located at the system call level.

#### Design Principles

The central aim of a predictable execution of applications under Osprey is supported through the following aspects: The OS yields a predictable performance through partitioning of memory and kernel data. Isolation between applications is achieved through the concept of resource containers that describe application requirements and ensure that the defined bounds on resource usage are kept. Communication between applications is strictly based on messages. As Osprey is targeted on computing clouds, the network interface also plays an important role. Osprey aims to provide efficient networking through replication of protocol stacks over the applications.

#### Structure and Implementation

Similar to the multikernel approach in Barrelfish (see sect. 4.1.2), the kernel state and memory are partitioned in Osprey. Thus, the need for synchronisation is reduced. Also, each core executes its own scheduler independently. Applications may be migrated between cores by a load balancer, but they can also be pinned to cores. Clock and inter-core interrupts are only used for preemption of running processes. I/O interrupts are passed exclusively to dedicated *noise cores* to avoid influences on applications running on other cores. Interprocess

communication in Osprey is handled through asynchronous message queues. These are also used to implement system calls. To ease application development, a cooperative user-level threading library is available.

The process model and message queues are tightly coupled in Osprey, which is detailed in (Sacha et al. 2013): On user level, each (heavy-weight) process provides an address space for one application. A process consists of multiple (light-weight) threads that execute application code. These threads can run concurrently, but not in parallel. Inside the kernel, each process is mapped by a task. Each task and each thread has its own message queue. Additionally, Osprey provides a kernel-to-userspace (K2U) and a user-to-kernelspace (U2K) queue for each task/process pair. Message delivery is facilitated by two schedulers, one running inside the kernel, and another running as an additional thread inside each process. A thread sends a message by putting it into the U2K queue. The associated kernel task delivers messages from this queue to the incoming queue of the kernel scheduler on the target core, which in turn forwards the message to appropriate task. The task sends the message to its userspace process through its K2U queue. The thread scheduler of the process delivers messages from the K2U queue to the appropriate tasks. To speed up messaging, Osprey also provides fast-path messages through direct queues between communication partners.

Osprey provides isolation of applications through *resource containers*. A resource container describes an application's requirements concerning CPU time, maximum event delays on external events, or memory budget. Osprey distinguishes preemptive and non-preemptive resources. Preemptive resources, like CPU time or network bandwidth can be revoked by the kernel at any time. A resource container specifies the minimum amount of these resources that is available to an application. Non-preemptive resources like memory cannot easily be revoked and thus are assigned for the resource container's lifetime. Resource containers specify a maximum amount they will use for these resources. The real-time scheduler of Osprey allows time-division multiplexing as well as assigning cores to single applications exclusively. Besides, a resource container can provide a private namespace that translates between application-local names and those used by the kernel, thus increasing portability of applications. The security of the system is enhanced through isolation of namespaces.

Osprey facilitates backward compatibility for legacy applications in three ways: (1) An unmodified binary application can be associated with a *buddy operating system* process that handles the application's system traps. (2) The source code of the application can be re-linked with a library operating system that implements the original OS interface used by the application and translates those system calls to Osprey system calls. (3) The implementation of runtime environments like Python, Java or POSIX allows to run unmodified legacy applications.

The network protocol stacks in Osprey run in user space to achieve low communication latencies, maximum throughput, and mobility events (application migration, network address changes). Additionally, each application has its own protocol stack (Sacha and Mullender 2012). The kernel only demultiplexes incoming packets and dispatches them to the user applications. The kernel also handles serialisation of outgoing packets, thereby heeding the resource constraints specified in the resource containers.

A checkpoint and restart mechanism improves fault tolerance and can be used to migrate applications between machines or cores. This mechanism is eased by the namespaces decoupling the application's interface from the real hardware, and the fact, that the biggest part of an application's state is stored in user-space.

#### Discussion

So far, Osprey is the only OS that explicitly targets real-time systems. The partitioning of the kernel and the concept of resource containers support OSR-1 (Predictability and analysability) and OSR-2 (Partitioning). OSR-3 (Communication) is fulfiled through Osprey's IPC mechanism. The load balancing mechanism might also be used for OSR-4 (Reconfiguration). However, no information is available on the code size of Osprey, and how kernel data is organised.

## 4.2. Virtualisation for Real-Time Systems

The use of virtualisation (Popek and Goldberg 1974) in SCSs allows to integrate applications with possibly different requirements and/or criticality in single computers (see e.g. Heiser 2008; Heiser 2011). The partitioning concepts implemented by a hypervisor ensure freedom from interferences between the partitions/applications. Thus, the consolidation of ECUs is eased, as the applications formerly distributed over multiple ECUs can each be provided with its own virtual ECU.

Industrial solutions for virtualisation in SCSs have been available for several years, e.g. the *Integrity Multivisor* (Green Hills n.d.), or the *LynxSecure Seperation Kernel Hypervisor* (Lynx n.d.). Such solutions are usually certified against standards for their use in safety-critical environments. Another example is PikeOS (Kaiser and Wagner 2007), which is based on the L4 microkernel principles (Liedtke 1995). Here, it is also important to note that the distinction between microkernels and hypervisors is blurred in places, as they are based on similar concepts and follow similar goals (Hand et al. 2005; Heiser et al. 2006). In the following, a brief overview of research on virtualisation for embedded/safety-critical system is given.

Like industrial solutions, the research hypervisor XtratuM (Masmano et al. 2009; Crespo et al. 2010) implements paravirtualisation. This concept requires that the code of guest OSs or applications is adjusted. With XtratuM, the use of hypervisors for SCSs is investigated. It is designed using the ARINC-653 concepts as a reference and provides temporal and spatial isolation to virtualised OSs and bare applications in partitions. Fault management techniques help to ensure proper operation of an SCS even in the presence of faults. The original implementation was performed on a LEON2 processor. Later, XtratuM was redesigned for the implementation on a LEON4 multicore processor (Carrascosa et al. 2014).

Bruns et al. (Bruns et al. 2013) push the concept of virtualisation in embedded systems further by considering systems without MMU. They use paravirtualisation and link partitions to separate address regions. Time partitioning is provided through cyclic scheduling. The concept is evaluated on a ARM Cortex-M3 microcontroller, where the hypervisor adds an overhead of 7 kB ROM and 1 kB RAM.

Baldin et al. aim to overcome a central drawback of paravirtualisation, namely the need to adjust the code of a guest OS. With *Proteus* (Baldin and Kerstan 2009), they propose a hybrid platform that can provide both full virtualisation and paravirtualisation as well. Using full virtualisation allows to run a guest OS without any adjustments to its source code. This can also help, e.g. during the consolidation of applications that are only available as binary code. Paravirtualisation can be use to achieve higher performance. An evaluation of Proteus was performed on a PowerPC405, showing both its performance and its small memory footprint ($< 11$ kB code and data).

The *NOVA microhypervisor* (Steinberg and Kauer 2010) is designed as a minimal hypervisor. Its aim is to have as little code as possible running in privileged mode, thus having a minimal trusted computing base. The reduced attack surface improves the security of the system. A core concept of NOVA is the implementation of virtualisation at user level: Only the hypervisor runs in privileged mode, while multiple VMMs can be run in user mode.

In their work on *RT-Xen*, Xi et al. (Xi et al. 2011) investigate the application of real-time scheduling theory to hypervisors. They implement several fixed-priority servers in the Xen hypervisor (Barham et al. 2003).

Summed up, virtualisation is important for SCS, as it enables the deployment of different applications with possibly different requirements on a single computer. Interferences between the applications are prevented through partitioning in time and space provided by a hypervisor. Thus, it is also possible to migrate legacy applications to new platforms without the need for extensive re-engineering of their code. Different application requirements concerning the underlying OS are accommodated through the fact that the OS is executed inside virtual machines.

## 4.3. Conclusions

Existing approaches for manycore operating systems mainly target domains like general-purpose or cloud computing. Such systems require a great flexibility as they must be able to execute applications with different requirements which are not necessarily known at design time. In comparison, SCSs today are configured statically and do not require techniques like dynamic resource allocation. Nevertheless, existing approaches cannot per se be excluded from safety-critical domains, as most of them leave a great flexibility in their implementation. Obviously, some mechanism, like dynamic name resolution can hardly be implemented in a manner that is feasible for SCSs. However, basic capabilities in this area may become relevant in the future, at least for a dynamic reconfiguration of SCSs (Pagetti et al. 2012).

For other mechanisms, solutions may be available that meet the requirements of SCSs. There are also commonality to be found in most presented OS approaches. Even though not targeted at SCSs, the concept of partitioning is often used to improve the performance of systems. Communication is usually based on explicit messages, which are easier to analyse. Finally, many approaches move functionalities out of the kernel to userspace, e.g. for scheduling of processes or application threads. Some of the available approaches appear to be fit better for their use in SCSs than others. Based on the above discussions, the fos approach is chosen as the base for this work. It fulfils the basic requirements from sections 2.3 and 3.4 to the highest degree, and simultaneously allows for a high degree of flexibility concerning the implementation of an actual SCS.

Virtualisation in SCSs is used as a vehicle to integrate applications with different requirements on single computers. Compared to full-featured OSs, the level on which partitioning is provided lies much lower. Thus, it is also possible integrate different OSs inside the partitions. Although not being at the center of this work, the concept of partitioning will be addressed where necessary.

# Part II.

# An Operating System and Selected Aspects

# 5 The MOSSCA Approach

*This chapter is based on Kluge, Triquet, et al. 2012; Kluge, Gerdes, and Ungerer 2014a*

This chapter presents MOSSCA, a *manycore operating system for safety-critical applications*. As discussed in chapter 3, a manycore processor is similar to a distributed system. An operating system should heed this resemblance (Baumann, Peter, et al. 2009). MOSSCA is founded on the concepts laid down by the works on fos (see sect. 4.1.3. Special heed is taken to make the general MOSSCA concepts amenable to analyses required in SCSs. Not all aspects of an OS can be determined on the level of these general concepts, but depend on an actual implementation. Insofar, MOSSCA serves also as an evaluation platform for concrete OS mechanisms that are discussed in the subsequent chapters of this part.

The following section 5.1 defines the hardware architecture that is underlying the MOSSCA reference implementation. The abstractions that MOSSCA provides to developers are introduced in section 5.2. The general MOSSCA architecture is presented in section 5.3. Section 5.4 presents the principles that underlie the MOSSCA prototype implementation. The MOSSCA concepts have already been applied to concrete domains and actual hardware platforms. These use-cases are presented in section 5.5. The fulfilment of SCS requirements by the general MOSSCA architecture is discussed in section 5.6. Section 5.8 surveys the analysis of a MOSSCA system more closely and highlights specific OS mechanisms that are examined in subsequent chapters. Section 5.9 closes this chapter with a brief summary.

## 5.1. Assumed Hardware Architecture

MOSSCA builds on the homogeneous manycore architecture that is laid down by Metzlaff et al. (Metzlaff et al. 2011). This architecture is aimed to be amenable for the use in hard real-time systems. It is based on the following principles:

**Simple cores** The cores of the manycore processors are as simple as possible, relinquishing the use of complexity-inducing hardware concepts such as branch prediction or out-of-order execution. This simplicity facilitates the WCET analysis and allows to derive tight WCET bounds. Furthermore, each single core requires only a small amount of space, which allows to integrated even more cores on a single chip. High performance can be achieved through diligent parallelisation of applications.

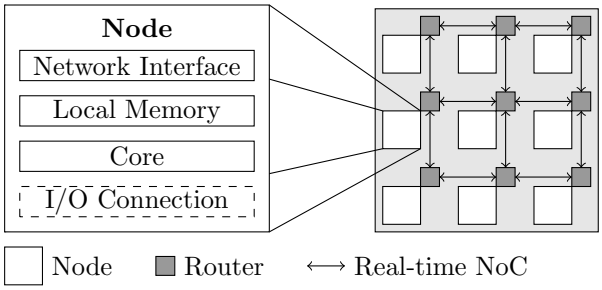



Figure 5.1.: Manycore architecture

**Local memories** Each core possesses local memories for code and data. Complex cache hierarchies that would impede a tight WCET analysis thus are abolished. Parts of the local memory may also be used as software-managed scratchpads. These can achieve a throughput similar to caches (Udayakumaran et al. 2006), but provide a better predictability (Wehmeyer and Marwedel 2005; Suhendra et al. 2005; Metzlaff and Ungerer 2014). If the amount of local memory available at each core is not sufficient for some applications, memory may be borrowed from other cores using NoC messages to access the memory. Additionally, the type of accessed memory can be encoded directly in the load/store instructions to make it visible to the WCET analysis (Schoeberl et al. 2011). In the future, it may also be expected that accesses to larger amounts of cheap (in terms of access time) memories can be made available through 3D stacking techniques (Black et al. 2006; Loh 2008). It is assumed that the processor does not possess a global address space shared among all cores. The local memory on any node is only accessible from the same node.

**Static hard real-time NoC** The NoC provides a predictable timing behaviour. Hard real-time communication on the NoC is scheduled statically. The NoC may offer additional dynamic scheduling for best effort messages (see e.g. Bjerregaard and Sparso 2005; K. Goossens et al. 2005; Stefan et al. 2012), as long as hard real-time traffic is not influenced. Messages are sent resp. received explicitly, e.g. through special instructions.

The architecture resulting from these principles is depicted in figure 5.1. The manycore processor consists of several nodes that are connected by real-time NoC. The main parts of a node are its local memory and core. It is connected to its NoC router by a network interface. I/O pins may be connected to the NoC, or they may be connected directly to single nodes as indicated in the figure. These principles mainly help to fulfil OSR-1 (Predictability and analysability). Fast, fine-grained communication (OSR-3 (Communication)) is achieved either by executing concerned applications on the same node, or by using the mechanisms provided by the NoC. To accommodate for OSR-2 (Partitioning), additional mechanisms are required:

**Execution modes** Each core shall come with at least two execution modes, usually called *user mode* and *supervisor mode*. Code that is executed in supervisor mode has full access to the complete core. User mode code has only restricted privileges and may not, e.g. change certain configuration registers of the core.

**Memory protection** It shall be possible to mark only certain regions of the memory to be accessible by user mode code. Access privileges are usually classified as *read*, *write*,

and *execute* privileges. Configuration of the memory protection mechanism can only be performed by code executed in supervisor mode. Memory protection can be implemented either using sets of *range registers*, or based on the pages managed by a MMU, whereupon the former approach is rather in line with the requirement for simple cores.

On software level, these mechanisms allow the implementation of *kernel mode* (executed in supervisor mode) and *user mode* code. Kernel mode code has full access to hardware units of the node and thus can configure a container for user mode code. This is mainly done by defining the memory regions that user mode code is granted access. Thus, malfunctions inside user mode code are restricted to the scope it is executed and cannot influence other parts of the system.

A further mechanism from which a SCS may profit is the concept of *privileged messages*. Such messages shall be used to control the execution on a node from a remote node. A special case of this concept can be found in the Parallella manycore (Adapteva 2013c): The core's control registers are mapped into the global address space, thus allowing other cores or even code executed on an external processor to start and stop execution of a single core. Message-based communication in today's manycore processors is usually based on a global address space. Memory protection mechanisms will be helpful to differentiate between privileged and unprivileged messages and to avoid unpredictable or unwanted interferences between applications.

The hardware concepts defined above may seem restrictive compared to currently existing manycore processor. Nevertheless, an OS architecture developed under these principles can also be applied to a less restrictive hardware architecture. Concerning the complexity of cores, more powerful microarchitectures do not inflict the performance of the OS, but they may pose a problem for WCET analysis. Also, dealing with a heterogeneous processor architecture is possible, as the works on the Barrelfish OS show (see sect. 4.1.2). Some of today's manycore processors possess only caches instead of full local memories. By employing cache partitioning and preloading mechanisms, it is possible to use these like local memories. Finally, works on processors with no real-time capable NoCs show that even on such processors a predictable application execution is feasible, even though performance losses may be expected (D'Ausbourg et al. 2011).

## 5.2. MOSSCA Abstractions

The basic MOSSCA approach assumes that enough nodes are available to execute each application thread on a separate node. Thus mutual influences between partitions that stem from pure thread execution are removed. By assigning threads of the same application different cores, even interferences inside an application are reduced. Threads belonging to the same partition can be grouped together by mapping them on adjacent cores. This will be helpful especially in processors without a real-time NoC to keep communication over the NoC as local as possible and thus reduce interferences. Communication between threads in one partition thus mostly takes place on the part of the NoC that is covered by this partition. It is strictly based on passing messages between the threads. Space and time partitioning thus is partly provided by the underlying hardware. This approach results in a mapping as depicted in figure 5.2. To make these properties usable by the application developer, MOSSCA provides three abstraction concepts, namely *nodes*, *communication channels*, and *servers*.

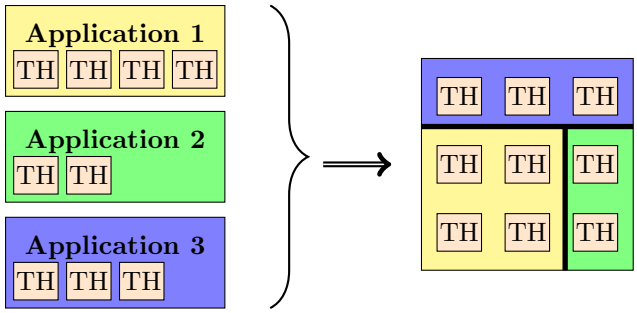



Figure 5.2.: Mapping of threads (TH) and partitions to a manycore processor

### 5.2.1. Nodes

Nodes represent the primary execution resources for applications. Each MOSSCA node maps directly to a physical node of the manycore processor. The presence of functional units in the physical node is represented by *node capabilities*. Such capabilities can, e.g. stand for timer units or caches. Another kind of capability are I/O pins that are directly connected to the node. A MOSSCA node acts as container for the binary image of one application thread and executes the program defined in this image, i.e. one node executes one thread. If the thread uses only node-local resources it is executed in full isolation from the rest of the processor and thus is not subject to any interferences from other threads. An initial configuration of the physical nodes is performed by MOSSCA during the boot process, where predefined application threads are loaded to the nodes. To allow for the upcoming need for higher flexibility, nodes may also be reconfigured during runtime, e.g. to execute a different application.

### 5.2.2. Communication Channels

*Communication channels* represent the basic means for interaction between threads. A communication channel in MOSSCA provides unidirectional communication between a sender *(channel source)* and a receiver node *(channel sink)*. The channel provides certain properties and constraints to its source and sink nodes. These properties are defined in the *channel policy*. The channel implementation ensures that an application cannot exceed the limits defined in the channel policy. Thus, possible overload of the NoC and the channel sink is prevented. If an application thread is trying to exceed the limits stated in the channel policies, the messages are blocked by the kernel, as the application is behaving outside its specification. Thus, MOSSCA ensures that faults in one application threads cannot propagate through the system.

Channels can be used to implement more sophisticated communication patterns on top. Two-way communication is achieved by coupling two channels with opposite directions. With the help of servers (see next section), it is also possibly to implement distributed buffers or blackboards, or other forms of group communication e.g. for inter-partition communication.

### 5.2.3. Servers

MOSSCA servers provide services that are used by multiple applications or threads but need to be executed in a centralised manner. An obvious example is an I/O server that multiplexes

I/O requests from several threads on one I/O device, and works as the primary interrupt handler for this device. Similarly, certain OS services that may afflict other threads need a central instance that has a global view of the system and thus is able to ensure isolation properties. Servers can also be used to coordinate inter-partition communication. Finally, applications may define servers that provide library services for computations used in several threads. If these are centralised, memory is saved in the nodes that execute the application threads. Similar to communication channels, MOSSCA servers must provide a predictable behaviour to their clients. This means that they must guarantee a maximum reaction latency that is defined in the *server policy*. The server policy also implicitly controls the actual communication between server and client application over the NoC. MOSSCA servers implement non-interruptable transaction. Once a server has started processing an application request, it will finish this request without interruption by other application requests.

### 5.2.4. Interface

Figure 5.3 illustrates a developer's view of a MOSSCA system and how the abstractions introduced above are used. The figure shows a MOSSCA system consisting of two applications and two servers. Application A consists of three threads A0, A1, A2, each executing on its own node N0, N1, N2. Application B has two threads B0, B1 executing on nodes N4, N5. Each server is also assigned a separate node, whereby the I/O server gets node N6 that possesses additional I/O pins. Using MOSSCA channels, the threads of one application can exchange data. Communication between the application partitions is handled by an inter-partition communication server running on node N3. The connection between application threads and servers is not mapped by (user-level) channels. From application view, only the server policy is relevant for the usage of the server.

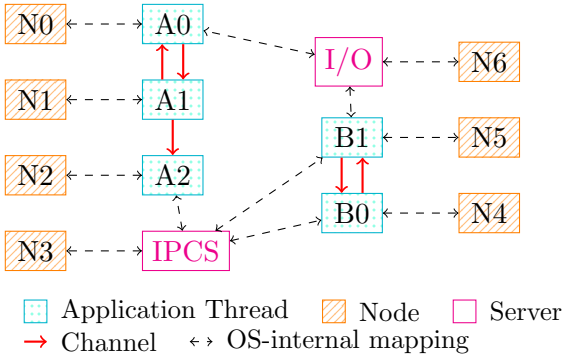



Figure 5.3.: The developer's view on a MOSSCA system

## 5.3. MOSSCA System Architecture

The MOSSCA architecture is based on the core idea of the factored operating system (Wentzlaff and Agarwal 2009) (see section 4.1.3) of distributing functionalities over the manycore chip as far as possible. Therefore, application threads are mapped to separate nodes. Additionally, MOSSCA allocates separate nodes as servers to perform tasks that must be executed in a centralized manner. These include off-chip I/O (for all partitions) and inter-partition

communication, but also application specific services. These concepts are reflected by the architecture of a MOSSCA system as depicted in figure 5.4 (Kluge, Triquet, et al. 2012; Kluge, Gerdes, and Ungerer 2014a).

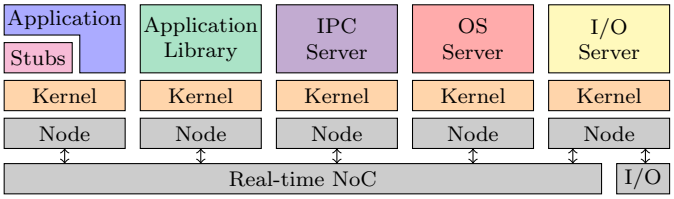



Figure 5.4.: MOSSCA System Architecture

The hardware base is formed by *nodes* that are connected by a *real-time interconnect*, as defined in section 5.1. Additionally, some nodes may have special facilities, e.g. for off-chip I/O. OS functionalities are split into several parts to run in a distributed manner and to achieve high parallelism. An identical kernel on each node is responsible for configuration and management of the node's hard- and software. OS functionalities that require a centralised execution or need to coordinate between several nodes are provided by OS servers. Off-chip communication is managed by dedicated I/O servers. For inter-partition communication, MOSSCA allocates dedicated communication servers (inter-partition communication server, IPCS). Additionally, MOSSCA supports the implementation of application-specific servers that provide e.g. library services used by several threads.

### 5.3.1. Kernel

The kernel manages all node-local hardware devices. Concerning e.g. the interface to the NoC interconnect, this includes a fine-grained configuration of the send/receive bandwidth within the bounds that are given by the OS. The kernel ensures that the local application does not exceed these constraints. This is especially important when the manycore hardware does not provide a real-time NoC to prevent a misbehaving node from flooding the NoC. The kernel is also responsible for the software configuration of the node, i.e. loading the code of the application-level code that should run on the node, including OS, I/O and other servers. To achieve these aims, the kernel can interact with an OS server.

### 5.3.2. Servers

Servers in MOSSCA provide services that cannot or should not be executed on every node. Services that cannot be executed on each node typically concern management of, or access to shared resources like I/O devices. If multiple application threads use the same library routine, it can be beneficial to execute this routine on only one node acting as *Application Library Server* to save memory on the other nodes. Depending on their functionality, servers may be used by multiple partitions. Dedicated OS servers manage the on-chip resources of the manycore processor. In contrast to other servers, they are cannot be addressed directly by applications. Instead, a local kernels may interact with an OS server if required.

### 5.3.3. Stub Interfaces

Applications can use a generic server interface to communicate with all servers except the OS server. However, server requests have to follow a certain protocol that is defined by the server developer. To ease the use of any server, server developers are encouraged to provide an additional server stub component that provides a server-specific interface. Internally, the stub maps to MOSSCA's generic server interface.

### 5.3.4. Generality of Approach

The MOSSCA architecture as defined in this section is very flexible and may be applied to a wide range of actual manycore processors. Although it is based on concrete hardware properties as defined in section 5.1, these properties themselves are so general that they can at least be emulated in actual manycore processors. In the following paragraphs give some examples to illustrate the generality of the MOSSCA approach.

First, consider communication: The assumed hardware provides explicit messaging for exchanging data between node. MOSSCA reflects this by exposing the messaging nature of the underlying hardware towards the application developer. However, MOSSCA messaging does not necessarily require that the hardware provides means for explicit messaging. Instead, messaging can also be implemented using shared memories or address spaces. An example for such an approach can be found in the eSDK (Adapteva 2013d) for the Parallella computer (Adapteva 2013c).

Closely related to communication is the synchronisation of threads. In MOSSCA, basic synchronisation is ensured through use of explicit messages, similar to the message passing interface (MPI 2012). If multiple threads need to operate on a resource that physically cannot be shared, e.g. an I/O port of the processor, this resource is managed by a MOSSCA server that multiplexes requests (arriving in the form of messages) from multiple threads.

As a last example, consider space and time partitioning between multiple applications running on a single manycore computer. Again, the assumed hardware provides space partitioning by having local memories inside each node that cannot be accessed from other nodes. If the processor has an address space that is shared among multiple or all cores, then means for memory protection must be used. Time partitioning is initially achieved by assigning each thread its own node. Thus, no interferences stemming from pure sequential execution can occur. If a thread uses a server which may be shared with other applications, then appropriate arbitration resp. scheduling of requests inside the server must ensure a predictable timing behaviour. Insofar, requests to servers are similar to critical section protected by mutex variables in a shared memory system, and require also a similar analysis.

A comprehensive example for the generality of the MOSSCA approach can be found in section 5.5.3 which discusses the implementation of MOSSCA on a shared-memory manycore processor.

## 5.4. Reference Implementation

A reference implementation of MOSSCA has been performed on the MacSim manycore simulator. The simulator is based on the hardware principles defined by Metzlaff et al. (Metzlaff et al. 2011) and implements an architecture similar to the one defined in section 5.1. Communication between cores in the simulator is purely based on messages, the cores do not share

a common address space. This section presents the concepts underlying this implementation and how it realises the MOSSCA abstractions.

### 5.4.1. Basic Principles

The following principles underlie the whole implementation:

#### Server API and OS Interface

Applications can send requests to servers using the generic API services defined for server usage. While the API services define a function interface, the actual request is performed through messages sent over the NoC. It would also be possible to expose the messaging nature of server requests explicitly to applications, thus reaching a higher grade of transparency. However, this approach was not taken to keep the MOSSCA API uniform. To make any concrete server implementation easier to use, it is possible to implement user-level libraries to wrap the generic server API calls into more device-specific functions.

Internally, a server must implement an interface for control and management through the OS. Over this interface, service requests from application nodes are handed to the server and replies are sent to the applications. The interface is also used by the OS to mediate usage permissions requested by applications. In return, the OS provides a framework to ease the implementation of servers. The framework includes managed buffers for messages to and from the server.

#### Messages

All communication in MOSSCA is based on messages that are sent resp. received explicitly. Concerning application messages sent through channels, these messages stay in some buffer in the receiver node until they are explicitly read by software. While being sensible for user-level messages, this approach does not suffice for OS-internal control. If, e.g. the OS needs to stop a misbehaving application, it needs to directly address the kernel on the appropriate node. In turn, the kernel should react fast and interrupt application execution. To achieve such aims, MOSSCA employs so-called *privileged messages* (see section 5.1) to address the kernel of a node directly. Instead of simply being put into a buffer upon arrival, privileged messages additionally trigger an interrupt and thus can be processed with low latency by the kernel. In principle, such privileged messages can impact the predictability of application execution. They must be carefully used during proper execution. However, if used in exceptional situations, system predictability may already be harmed. In such a case, the system will actually benefit from such messages as they allow to restore a valid system state with low latencies. In MOSSCA, only OS servers are able to send such messages.

#### Server Scheduling

In the basic implementation, all MOSSCA servers process requests in first-in first-out (FIFO) ordering. This allows to bound the response time for any request, depending on the number of clients that use a server. However, in this approach the bound incurs a high pessimism due to the assumption that any request has to wait for requests of all other clients to be finished. Less pessimistic bounds may be derived by performing a schedulability analysis and/or by using more sophisticated scheduling schemes.

### 5.4.2. Kernel

The kernel is MOSSCA's component that is in closest touch with the applications. It provides the full MOSSCA API through its system call interface. An additional messaging interface is used for coordination with an OS server.

#### System Call Interface

The kernel's system call interface provides the MOSSCA API. From kernel view, this API can be divided in two groups: (1) services that need interaction with an OS server, and (2) services that can be performed locally or in concert with a non-OS server. Locally performed services are use of channels (send/recv). Sending and receiving through/from a channel requires only interaction with the node's local hardware. For sending over a channel, the kernel plays an important role in ensuring that the defined usage bounds are kept by the calling application. This is important if the real-time capability of the NoC can only be ensured through additional flow management performed in software. Insofar, the kernel of a sender node also ensures that the receiver node will not experience an overload situation. Requests to regular (non-OS) servers are handled in a similar manner. Like for sending over channels, the requester's kernel controls the traffic the server experiences. API services for resource de-/allocation or that concern execution control, e.g. activation and termination of threads, need coordination through an OS server and possibly interaction with the kernel on some other node. The use of the OS server is transparent for the application from a functional point of view. Nevertheless, the calling application will experience longer latencies compared to system calls that are performed locally.

#### Messaging Interface

The messaging interface of the kernel is used for configuration and control through the OS server. This interface provides means to load application code on the node and to control application execution, i.e. starting and stopping of applications. These services can only be invoked by an OS server. The usage policies of channels and servers used by the local application are also set through the kernel's messaging interface.

#### Structure and Mechanisms

The MOSSCA kernel consists of two components, namely *communication control* and *execution control*. Communication control is responsible for any kind of communication originating in the application, i.e. it handles *channel* and *server* abstractions used by the application. For allocation of these abstraction and setting of usage policies, the kernel interacts with the OS server. Fine-grained control of usage of these resources is solely performed by the kernel. *Execution control* takes care of loading an application image and running the application thread. If requested by the OS server, it can also interrupt a running application thread. For API services requiring interaction with an OS server, the kernel performs as much work as possible locally. Only work that needs interaction with other nodes or a global knowledge is marshaled into messages. These are sent to an OS server, where they are processed. All mechanisms implemented in the kernel deliver a predictable timing behaviour.

If several application threads have to be executed on one node, a user-level scheduler (Anderson et al. 1992) can be employed by the application developer. Application threads

running on the same node should belong to the same application. Thus, the scheduler can be simplified compared to a full-featured OS scheduler as it does not need to cross addressing space boundaries. In this scenario, the application has full control of the execution resource. System analysis is not complicated by an OS scheduler and interrupts.

### 5.4.3. OS Server

The OS server manages all on-chip resources, and especially the mapping and scheduling of application threads to nodes. Which thread is executed on which node can either be decided by an OS server dynamically, or already during system integration to give stronger guarantees. A MOSSCA system can have several OS servers running in parallel, each managing a part of the processor, or fulfiling different tasks. Thus, if e.g. each partition has its own OS server, interferences between partitions are kept at bay. With the help of the kernel, the OS servers manage the NoC interconnect and set up send/receive policies for all nodes. One OS server instance also coordinates the boot process of the chip and configures all other nodes by deciding which code to execute on which node. If the need for online-reconfiguration of the system arises, this task is also managed by an OS server. Applications cannot directly access an OS server. Instead, the kernel interacts with an OS server to implement certain system services. The kernel communicates with the OS server through messages. Therefore, the OS server possesses only a messaging interface.

#### Messaging Interface

The messaging interface of an OS server is subdivided in two parts: The first part is responsible for communication with kernels that need to coordinate with the OS server. Over this interface part, execution control of threads is performed and the resource allocation/free requests are transmitted. Support for a more dynamic system, e.g. dynamic allocation of nodes, can be added in this part of the interface. The second part of an OS server's messaging interface is responsible for coordination between multiple OS servers and with other servers. This is important if, e.g. new channels are allocated during runtime.

#### Structure and Mechanisms

Like in the kernel, mechanisms in an OS server can be divided in communication and execution control. However, the OS server performs these tasks on a more global level. The communication control part of the OS server manages channel and server connections. If such a new connection is requested, the OS server checks whether the required resources are available. For server usage requests, this work is performed in concert with the affected server. The execution control part of the OS server manages the execution resources of a part of the processor. Additionally, it performs some monitoring of the system to ensure proper operation. If it detects anomalies in application behaviour, it can interrupt execution of these applications. All OS server mechanisms are implemented as non-interruptable transaction to deliver a predictable timing behaviour to applications.

### 5.4.4. I/O Server

If multiple applications must share access to the same I/O resource, this access is mediated by an I/O servers. It is irrelevant whether the I/O resource can only be accessed from one node,

or via the NoC by any node. An I/O server takes care of time multiplexing the applications' I/O requests towards its assigned I/O device. It provides means for bandwidth and latency management which allow to give certain timing guarantees to applications. The concrete implementation of the management depends on the specific device. MOSSCA provides general helpers to alleviate this implementation. The I/O Server is also the primary interrupt handler for its device. If necessary, it may forward interrupt requests to specialised nodes.

### 5.4.5. Inter-Partition Communication Server

For communication between partitions, MOSSCA can provide an inter-partition communication server (IPCS). While channels provide a reasonable abstraction for fine-grained communication, more coarse-grained approaches that need further kinds of control can be implemented in such a server. The communication channel abstraction of MOSSCA can principally ensure a predictable communication between any two application nodes. Hence, it could also be used to implement communication between two partitions. Nevertheless, inserting an inter-partition communication server as mediator for coarse-grained communication can improve system reliability and composability. Received channel messages are stored in some buffer at the receiver node, either by the NoC interface, or by some software handler, until they are explicitly received by the application. Naturally, the space available for buffering is limited. An overflowing receive buffer may seriously impact the behaviour of the application thread running on the correspondent node. For communication within a partition, this would present only a minor problem, if at all, as the application developer should be able to tune and synchronise his own application threads appropriately. However, he might have only few knowledge about the behaviour of other applications he needs to interact with. Thus, if a message from another partition arrives at some node, the node might not be able to handle it immediately. Hence, this message will block buffer space and thus can influence the receive behaviour experienced by intra-partition messages arriving later. MOSSCA solves this problem through a dedicated Inter-Partition Communication Server. This server stores inter-partition messages until the receiver is actually ready to receive them. Thus, no inter-partition message can interfere with intra-partition messages. At the moment, the IPCS implements communication buffers and blackboards for unbuffered communication. Applications can access an IPCS through the regular API services defined for servers, or a user-level library wrapper.

If MOSSCA is deployed to an architecture where node-local memories are accessible by all other nodes through a global address space, no IPCS is necessary. Instead, buffer memory for messages should be allocated statically on the receiving nodes. An appropriate messaging software library must ensure correct use of these buffers.

### 5.4.6. Bootrom Server

The Bootrom server (`bootrom`) is a special kind of server that coordinates bootstrapping of a MOSSCA system. It is started on a dedicated node when the computer is switched on. `bootrom` is responsible for configuring the whole system: it ensures that the MOSSCA kernel is started on all nodes and subsequently loads configuration data and application images from persistent memory. The Bootrom server is discussed in more detail in chapter 6.

### 5.4.7. Construction of a MOSSCA System

From the viewpoint of MOSSCA, an application is mainly characterised by it its usage of channels and servers. In the application code, these channels and servers are represented by symbolic constants. An XML file is used to describe each application's interface *(application description)*. During system generation, dependencies between applications and/or servers are resolved and the symbolic constants are replaced through C macros. A MOSSCA system is described through a *system description* XML file. The system description contains information about the static assignment of applications to nodes. Furthermore, it describes the channel connections between applications and the server usage of applications. For all channels and servers, usage policies can be specified additionally. A *system generator* processes the information specified in the system and application description files. It generates build information for all MOSSCA components and all applications, e.g. the macros used to address communication channels. Additionally, it generates system and node configuration data that is used by the Bootrom server to initialise a MOSSCA system.

## 5.5. Use-Case Implementations

The MOSSCA concepts and its actual implementation have been employed in several projects. They are presented in this chapter. A first study shows the applicability of the MOSSCA concepts when porting AUTOSAR OS to a message passing manycore (sect. 5.5.1). In the EC project parMERASA, the system architecture used by three different application domains is initially based on the MOSSCA concepts (sect. 5.5.2). Finally, MOSSCA has been successfully ported to the Patmos platform of the EC project T-Crest (sect. 5.5.3).

### 5.5.1. AUTOSAR OS on a Manycore Processor

A study examined how parts of AUTOSAR OS (AUTOSAR 2011) could be ported to a message-passing manycore processor using the concepts of the MOSSCA architecture (Kluge, Gerdes, and Ungerer 2012). The study considered the parts of AUTOSAR OS that are inherited mainly from OSEK OS (OSEK Group 2005), and the multicore-specific parts.

#### Software Architecture

The AUTOSAR OS specification (AUTOSAR 2011) requires for multicores that most of the AUTOSAR basic software (BSW) is executed on only one core (called *master core*), but together with application code. Other cores can execute only a limited set of BSW services. If an application running on such a core has to use a basic software service, the call might be delayed by the applications running on the master core. AUTOSAR OS defines *OS Applications* as containers for tasks and associated OS objects like counters, schedule tables, etc. (AUTOSAR 2011). Figure 5.5 (left) shows two such applications, each consisting of three tasks; the OS objects are combined in the box labelled *OS*.

The MOSSCA concepts are applied in the following manner: Assuming that enough nodes are available, each task is assigned a separate node. Additional nodes are used for the management of OS Application objects (one node per application), BSW services, and I/O processing (actually part of BSW). A possible mapping of these components to a manycore processor is shown in the right part of figure 5.5. Thus, each AUTOSAR OS application is represented by a cluster of nodes. This means that only limited portions of code must be loaded

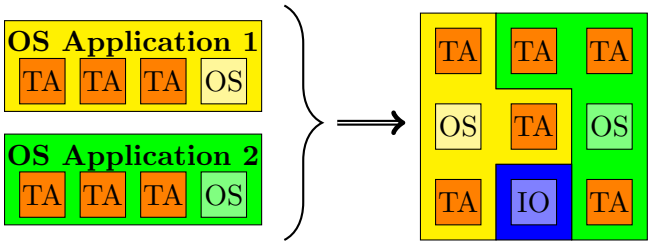



Figure 5.5.: Mapping of AUTOSAR applications to a message-passing multicore chip

to the cores respectively reside in the local memories. Insofar, the concepts of AUTOSAR OS are extended, which already achieve some locality by executing most of the AUTOSAR software stack on one core. The presented approach goes one step further by decoupling I/O operations, and by removing inter-application dependencies by segmenting the AUTOSAR OS and BSW between applications. Communication between the cores strictly follows the message-passing paradigm.

Figure 5.6 depicts the generic software architecture. A basic abstraction from the hardware is provided by a hardware abstraction layer (HAL) on each node. This could be an extension of the AUTOSAR *MicroController Abstraction Layer (MCAL)*. On task nodes, a small BSW wrapper provides the AUTOSAR functional interface. The wrapper translates the function calls into messages that are sent to the relevant servers. OS/BSW-Servers execute the AUTOSAR software stack except for external I/O, which is handled by the I/O servers.

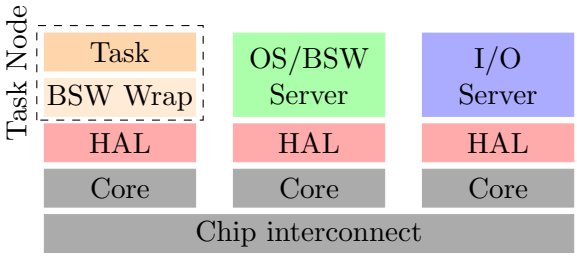



Figure 5.6.: Software architecture

### Implementation Concepts

The code on all nodes (task node, OS/BSW and I/O servers) is based on on loops. They are designed in a way such that waiting times of tasks calling OS/BSW or I/O services are bounded.

**Task Node** The basic structure of a task node is shown in algorithm 5.1. After system start, a task node waits for an activation message from its BSW server. If such a message arrives, it starts executing its task that was statically assigned during system integration. When the task has finished its work, it calls the `TerminateTask` service (line 4). Thereafter, the task node is ready for another activation.

Services provided by the OS/BSW and I/O servers are implemented as wrapper functions in the BSW Wrapper on a task node. These functions provide the interface specified by

**Algorithm 5.1:** Task node structure

```
1 while true do
2 |   Wait for activation message from BSW;
3 |   Task Execution;
4 |   TerminateTask;                          // part of task execution!
```

AUTOSAR. Internally, they send service request messages to the relevant servers and wait for reply. In the meantime, the calling task is blocked.

**I/O Server** The loop of an I/O server for processing tasks' I/O requests is kept quite simple (see alg. 5.2). The server waits for a message from any task. On receipt, the server processes this request and performs the I/O operation. Finally, it sends a reply message to the task and waits for the next request. Incoming requests can be processed either in FIFO order, or they can be reordered following client priorities.

**Algorithm 5.2:** I/O Server main loop

```
1 while true do
2 |   Wait and Receive I/O request from task;
3 |   Process request;
4 |   Perform I/O;
5 |   Send reply to task;
```

The I/O server is also responsible for handling interrupts originating from its associated I/O devices. So, at least parts of an interrupt service routines (ISRs) will be executed on the I/O server. Further interrupt processing must be performed by tasks running on other nodes to keep the I/O server's worst-case latency low.

**OS/BSW Server** Each OS/BSW server manages several task nodes that are assigned statically. The server holds all data that is relevant for managing the tasks, e.g. states and events of tasks. It is implemented similar to an I/O server, but may interact with multiple nodes during a single transaction. Algorithm 5.3 depicts the general functionality of the OS/BSW

**Algorithm 5.3:** OS/BSW Server main loop

```
1 while true do
2 |   Receive request;
3 |   Local message processing for reply;
4 |   Send reply;                          // if necessary/possible
5 |   Interaction/process to completion;          // if necessary
```

server loop. After receiving a service request, the server performs some local processing. Generally, this involves error checking and ensuring that the operation does not fail. For the OS services that were investigated so far, this work does not require any long-latency interaction with other nodes, as all OS and task management data is kept on the server. Thus, the server

can immediately send a return value to the task node (line 4). Just then, the server interacts with other nodes and process the request to completion.

Algorithm 5.4 demonstrates the functionality of the OS/BSW server (alg. 5.3, lines 3–5) for the `ActivateTask` system service as an example. First, the server has to check the activation conditions for task `t`. This work does not require any further interaction, as the server fully keeps track of the tasks' states. Thus, it can directly send a reply to the caller. The following work does not involve the caller anymore. The server checks, whether task `t` is currently suspended. If so, it sends an activation message to the core task `t` resides on. Else, if task `t` is already running, the activation is queued to be sent later, after the current instance of task `t` has terminated.

**Algorithm 5.4:** Processing of ActivateTask system service

```
   input: Activation request for task t
1  Check activation conditions for t;
2  Send return value to caller;
3  if error condition then
4  |  Cancel processing;
5  if t is suspended then                                  // t not running
6  |  Send run command to node of t;
7  else
8  |  Queue task activation
```

**Restrictions** The semantics of the event management are completely preserved. The same holds for the services that are used for alarms. Task management required some changes. The `Schedule` system service no longer has any meaning, as a task holds its core exclusively. For the `GetTaskState` service, it is important to note that the return value may be inaccurate. The task, whose state is queried, might have terminated while the `GetTaskState` service was processed by the OS/BSW server. Finally, the `GetTaskID` service can be implemented without a server request.

For the resource management services `GetResource` and `ReleaseResource`, it is not possible to preserve the semantics. When dealing with real parallelism, the priority ceiling protocol defined for these services will not work properly (Kluge et al. 2009). AUTOSAR OS (AUTOSAR 2011) accommodates this fact by explicitly forbidding resource sharing across cores. Inter-core synchronisation is performed by using spinlocks in shared memory. In this study, shared memory is abandoned, so the spinlock approach will not work either. However, the MOSSCA approach removes partially the need for the resource concept, namely when it is used to synchronise accesses to shared I/O resources. Instead, I/O operations are performed by dedicated I/O servers which give timing guarantees. The synchronisation of accesses to shared data structures is discussed in the following section.

### Properties

**Synchronisation and Shared Memory** In a multitasking system like AUTOSAR it is sometimes necessary to synchronise accesses to shared resources, i.e. I/O devices and data structures in a shared memory. As stated above, the resource concept of AUTOSAR works only

within a singlethreaded processor. The AUTOSAR OS therefore defines spinlocks for inter-core synchronisation. However, these come with several drawbacks. They are deadlock-prone, and they can result in contention of the memory interface through spinning. Although spinlocks can be implemented such that waiting times can be bounded (Craig 1993; Gerdes et al. 2012), using them on a message-passing processor is not feasible.

Due to the message passing nature of the system, there is no longer any need to employ locks. I/O is handled by dedicated I/O servers that arbitrate all I/O requests on a processor. A similar approach can also be implemented for shared data structures, where these are managed by dedicated servers. When porting an application, accesses to shared data must be changed to special service requests.

Although the current AUTOSAR standard requires the existence of a shared address space, it also introduces the concept of message-passing between OS applications. Therefore it defines the *Inter-OS-Application Communicator (IOC)* as means for explicit communication. The IOC provides means for buffered (send/receive semantic) and unbuffered (read/write semantic) communication. So one way to solve the problem of shared data could be to extend the IOC for inter-task communication. This would require some re-thinking of existing program code, as the state of data must be kept consistent between tasks. Some clues on how to achieve such consistency are given by Baumann et al. (Baumann, Barham, et al. 2009). Another solution, targeting the shared memory problem, is discussed in the next section.

**Efficiency** Today's automotive ECUs sometimes comprise over hundred tasks. However, only few of these tasks are running concurrently. The OSEK OS specification explicitly limits the number of tasks that are not in suspended state to 16. Although increases of core numbers can be expected for the future, one has to assume that at least for some time the number of tasks will be higher than the number of cores available for task execution. Insofar, the presented approach still needs an extension towards multitasking on task nodes. This might also alleviate the problem of shared memory. It would be possible to put tightly interacting tasks onto the same node where they would share the node's local memory or cache. Then, however, core-local schedulers would have to be added to the implementation.

#### Lessons Learned

The case study on AUTOSAR OS shows the feasibility of implementing an run-time environment (RTE) interface through a server and an interface stub. However, it shows also that the RTE interface and application code should be rethought. Compared to an implementation on a single-core processor, certain RTE services will necessarily exhibit a different semantic or behaviour, or may even no longer make any sense. Also, it is important to keep in mind that the response time for RTE services most probably increases, as the RTE server is a shared resource used by multiple application cores.

### 5.5.2. System Software in the parMERASA Project

The central aim of the parMERASA project (Ungerer et al. 2013) is the parallelisation of industrial real-time applications for future manycore architectures. Applications from different domains are investigated, namely avionics, automotive, and construction machinery. Existing applications from these domains are parallelised for execution on a manycore simulator (with

up to 64 cores) developed also in the project. The simulated hardware provides a shared address space and supports the shared memory programming model.

In the beginning, the MOSSCA concepts constituted the baseline for the system software developed in the project. Due to the shared memory nature of the investigated application programs, most concepts of MOSSCA were abandoned during the development of the system software. The final system software (Bradatsch et al. 2013) resembles well-known stacked architectures. A generic HAL provides basic abstractions from the underlying hardware and eases the implementation of high-level functionalities for scheduling, protection, communication and synchronisation, and I/O. On top of this HAL, domain-specific RTEs are implemented.

The distributed nature of the system is investigated in the context of the automotive RTE with respect to service calls that have to cross core boundaries (Bradatsch et al. 2014). In general, OS functionalities are executed on that core on which they are requested by the application. Each core therefore has its own set of OS data. However, some functions, like the activation of tasks, affect also the OS data of other cores. This work compares two implementation concepts, using either shared memory and locks or messages for communication between the concerned cores. In the shared memory approach, most work of the service call is performed on the calling core. The affected core is notified just at the end such that the local OS can react to changes in its data structured, e.g. b performing a rescheduling. When using messages, the larger part of the work is performed by the affected core, as only itself is allowed to alter its data structures. Although the shared memory approach shows a better performance, the message-based implementation is obviously preferable from an SCS point of view. There, OS data structures are kept locally (similar to MOSSCA), and only the owner is allowed to access and change them. Thus, another level for ensuring safety and partitioning is available, as the affected core might be able to detect (and counteract) misbehaviour of the calling core.

### 5.5.3. MOSSCA on the T-CREST Manycore Platform

The central aim of the project was to speed up worst-case execution paths on manycore processors, and to ease WCET analysis (T-CREST 2013). Therefore, a time-predictable manycore platform was developed. The base of the T-CREST platform is formed by the hardware, consisting of the processor, NoC, and memory hierarchy. A LLVM-based compiler interacts with static WCET analysis tools (Puschner et al. 2013) to generate appropriate code for the platform.

#### The T-CREST Hardware Platform

The hardware platform is based on Patmos processor cores (Schoeberl et al. 2011) that are connected by a statically scheduled TDMA (Schoeberl et al. 2012) and possibly asynchronous self-timed (Kasapaki et al. 2013) NoC. The memory hierarchy consist of a global shared memory and core-local caches and scratchpad memories. Each core has data and instruction caches. An additional stack cache (Abbaspour et al. 2013) is used to speed up stack accesses and improve WCET analysis (Jordan et al. 2013). Instruction and data scratchpad memories with each up to 64 kB can further be used to speed up application execution. Each Patmos core is a dual-issue RISC VLIW processor that supports predicated execution of instructions (Schoeberl et al. 2011; Schoeberl et al. 2015). The Patmos instruction set architecture (ISA)

has four classes of load/store instructions. These classes specify which part of the memory hierarchy is accessed. Thus, static WCET analysis can directly recognise, whether a load/store instruction targets the global memory, stack memory, data cache or the local data scratchpad memory (SPM).

FPGA prototype implementations of the hardware platform are available (T-CREST 2014) for several FPGA platforms. For porting MOSSCA, the implementation for the Terasic DE2-115 development board was used. This implementation features four Patmos cores. The sizes of the local memories are shown in table 5.1. The SRAM of the development board with 2 MB is used as global memory. A dedicated core has access to off-chip facilities like UART. Bootstrapping of the system is managed via this core (boot node).

Table 5.1.: Local memory sizes of DE2-115 T-CREST platform

| Type | Size |
|---|---|
| Method cache | 8 kB |
| Data cache | 4 kB |
| Stack cache | 4 kB |
| Instruction SPM | 1 kB |
| Data SPM | 2 kB |

### MOSSCA on the T-CREST Platform

When porting MOSSCA to the T-CREST platform on this board, the memory hierarchy is the greatest challenge. MOSSCA is designed to run on a manycore platform with core-local memories without the need for a global address space. In contrast, the design of the T-CREST platform favours an approach where most code and data reside in the global memory and only critical parts are put into the SPMs. Due to the stack cache, stack memory always must reside in the global memory.

The MOSSCA BIOS is replaced by the bootcode of the T-CREST platform.

The limited size of the SPMs does not allow to put both the MOSSCA kernel and applications into the core-local memories. To leave the SPMs for more frequently used application code and data, the kernel is put into the global memory. As this memory can be accessed by all cores, there is no need to replicate the code section of the kernel. Instead, all cores execute code from the same memory region. Nevertheless, each core uses its own instance of the kernel. Therefore, the global memory is partitioned, and data sections and stacks of the kernel are replicated into these partitions. Kernel data is mapped into a data structure. Accesses to this structure are performed over a pointer that is calculated from the ID of the concerned core. There is still a special kernel used for the boot node that manages initialisation of the whole system.

MOSSCA's *channel* abstraction is implemented using the message passing library of the T-CREST platform. Messages can be passed directly between core-local data SPMs.

### Lessons Learned

This MOSSCA port shows how the MOSSCA concepts can be mapped to actual manycore processors, even if they require that most code and data are stored in a global shared memory.

The memory architecture of the T-CREST platform posed the greatest difficulties, as the it required adjustments of the linking and deployment process of MOSSCA. The MOSSCA port to the T-CREST platform provides the basis for the analysis of execution models that is discussed in section 9.1.

## 5.6. Fulfilment of Requirements

MOSSCA fulfils OSR-1 (Predictability and analysability) in two ways. First, the OS it designed and implemented in such a manner that it provides a predictable behaviour and can be analysed. Second, it urges application developers to modularise their application in a meaningful manner. Through the modularisation and by basing communication completely on explicit messages, the analysability of applications is improved. For example, WCET analysis needs only to consider local memory accesses.

Requirement OSR-2 (Partitioning) is partially supported by the assumed underlying hardware. Placing applications or even application threads on separate nodes provides a basic partitioning in space and avoids timing interferences that stem from pure execution. Communication between application threads is managed through channels that ensure a predictable behaviour and prohibit unpredictable interferences. Use of other shared resources is mediated through servers. For a static system, the worst-case response times (WCRTs) of a server can be calculated and thus provides a safe upper bound on possible timing interferences.

OSR-3 (Communication) requires the provision of means for coarse- and fine-grained communication. Both can be performed over the channel abstraction. To improve system reliability, MOSSCA additionally provides inter-partition communication servers for coarse-grained communication between applications/partitions.

OSR-4 (Reconfiguration) is supported through the modular structure of MOSSCA. An OS server may implement mechanisms that actually perform a reconfiguration of the system. However, care must be taken to ensure that the target state of a reconfiguration is still valid. This can be achieved by supplying a number of analysed valid configurations that can be chosen. For using fully dynamic reconfiguration during runtime, such guarantees cannot be given.

The actual code size of MOSSCA depends on the implementation and the amount of functionalities that must be included. Table 5.2 shows the sizes of code and data sections of the MOSSCA prototype implementation compiled for an ARMV6-M ISA. In total, a MOSSCA kernel needs $< 5\,kB$ of memory. Thus, OSR-5 (Code size) can be considered as being fulfilled.

Table 5.2.: Kernel memory usage (bytes) for ARMV6-M ISA (compiled with `-O0`, each kernel needs some additional stack space

| Kernel | `.text` | `.rodata` | `.data` | `.bss` |
|---|---|---|---|---|
| Boot Kernel | 4,552 | 112 | 16 | 56 |
| Standard Kernel | 4,560 | 80 | 0 | 52 |

Sharing of data inside the OS is completely avoided OSR-6 (Shared data). OS tasks are performed locally as far as possible based on local knowledge. If a global knowledge is necessary to perform a task, an OS server that has the data at its disposal takes over the task. Sharing of data on application level is aggravated by basing all communication on messages. Thus,

application developers are urged to distribute application in such a manner as to minimise communication.

## 5.7. MOSSCA and Virtualisation

When regarding virtualisation in the context of MOSSCA, it is important to consider why virtualisation is used in today's SCSs at all. An important driver for virtualisation in SCSs is the consolidation of applications into single computers. Virtualisation allows to migrate not only the actual application, but also the underlying RTE and OS stack that is used by an application. The different applications then share the same computer with its physical resources, like I/O devices or memory. Temporal and spatial partitioning provided by the hypervisor prohibit these applications from interfering with each other. Especially, high-critical applications cannot be impeded from applications with lower criticality.

In the context of MOSSCA, partitioning is provided by the underlying hardware and the MOSSCA abstractions. To also provide RTE/OS services used by concrete application, a paravirtualisation approach is conceivable, as has been examined in the AUTOSAR use-case implementation (see sect. 5.5.1). In a more general manner, this means that for each RTE resp. OS interface a server implements the relevant services. Applications need to be adjusted such that, instead of directly calling a service, they send a request to the relevant server, and wait for the server's reply. Figure 5.7 exemplarily shows the architecture of a MOSSCA system where a real-time and a general-purpose application share the same manycore processor. On each application node, a stub provides the well-known functional RTE interface to the application. The stub converts the function call into a service request that is sent to the corresponding RTE server. In the figure, the real-time domain is using the AUTOSAR RTE, while the general-purpose domain is represented by the POSIX interface. An RTE server can be shared by multiple application nodes of the same domain. I/O servers are shared between both domains. Real-time guarantees for usage of the I/O server are derived through appropriate scheduling of requests and a schedulability resp. response time analysis. Like in the general MOSSCA architecture, the MOSSCA (OS) server is responsible for management of the whole chip, while the RTE servers manage only the nodes that are part of their domain.

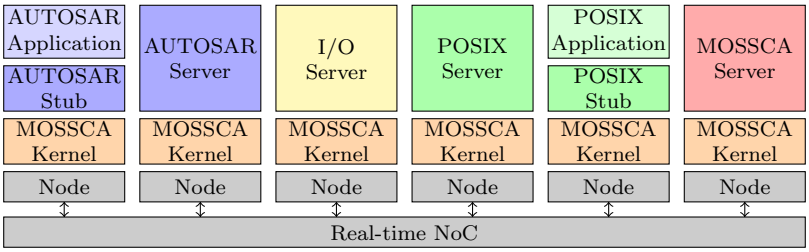



Figure 5.7.: Example architecture of a MOSSCA system providing paravirtualised AUTOSAR and POSIX RTEs

## 5.8. Analysis of a MOSSCA System

The analysis of a MOSSCA system must be carried out under two aspects. First, all relevant parts of the full architecture and their interactions among each other must be examined. The

analysis has to cover the whole stack of hardware, operating system, and application software of the generic MOSSCA architecture (cf. figure 5.4). The restriction to *relevant parts* means that e.g. application that are not safety-relevant can be ignored during the analysis as long as the OS ensures that they cannot interfere with safety-relevant parts of the system.

Second, the analysis must cover the whole life cycle of the system. Such a life cycle is schematically shown in figure 5.8. During any phase of the life cycle, proper operation according to the safety-critical requirements has to be ensured. The life cycle starts with a phase during which the system is switched *Off*, i.e. it is not operating. In this phase, the computer cannot influence the behaviour of the physical system, so other, physical means must be employed avoid any behaviour that may violate the safety property. When the system is switched on, it transitions into the *Operate* phase which itself is composed of three phases. During the *Bootstrap* phase, the system is initialised, i.e. the operating system and applications are started, and sensors and actors are possibly calibrated. When bootstrapping is finished, the actual *Operation* of the system starts. When the system is switched off, it passes through a *Shutdown* phase during which actors can be brought into a safe state and the software is stopped. Any part of the *Operate* phase may be interrupted by an error. Error handling is performed in an *Error* phase. Depending on the severity of the error, the system may be switched *Off*, or it transitions back into the *Operate* phase either by rebooting, or simply by continuing the interrupted phase.

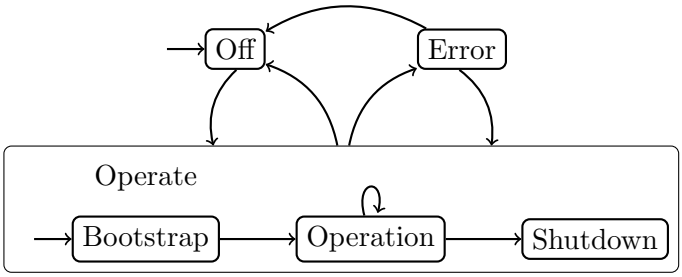



Figure 5.8.: Life cycle of a MOSSCA system (schematically)

It is a central requirement that operations on all phases heed their timing constraints. Therefore, the focus in this and the following chapters will be on the time-predictability of MOSSCA and applications during the phases. The next section, the boostrapping of a *mossca* system is discussed. Sections 5.8.2, 5.8.3 and 5.8.4 deal with the analysis of operations that are mainly used during the actual *Operation* of the system. Section 5.8.6 tackles the shutdown and error phases. Section 5.8.7 argues for the need to model timing properties in a detailed manner.

### 5.8.1. Bootstrapping

Most works on SCS ignore the boot process of a system and mainly deal with regular operation. As long as the physical system is in a safe state, boot strapping may have only low criticality. However, under certain circumstances it is necessary to give hard timing guarantees even for the boot process. This can be the case, for example, when a computer is restarted due to watchdog mechanism. As this can happen during regular operation, it is important to limit arising blackout times.

### 5.8.2. Scheduling of Server Requests

Any MOSSCA server is a resource that is shared among multiple tasks, possibly belonging to different applications. To enable a WCET analysis of applications, the worst-case waiting time (WCWT) for a server request must be determinable. The WCWT is influenced not only by the complexity of the actual service $W_S(c)$ and the NoC transmission times $W_T$, but also by number of clients $n$ that use a specific server. This number must be known, else the analysis would be infeasible. A very simple approach is to implement a server such that it handles incoming messages in FIFO order. It is important to keep in mind that a MOSSCA server processes requests as non-interruptable transactions, i.e. once the processing of a request has started, it has to be finished before the server can start processing another request. The WCWT $W(c)$ of a client that requests a service $c$ can be bounded by:

$$W(c) \leq 2W_T + W_N(c) + W_S(c) + (n-1)W_S(C) \tag{5.1}$$

The client experiences twice the NoC transmission latency $W_T$ for the request and the reply message. The time for client-side OS processing of the request is expressed by the term $W_N(c)$. Then the actual server-side processing time $W_S(c)$ is added. Finally, possible interferences from all other clients have to be heeded, assuming the worst case of each client requesting the most complex service $C$. It is obvious that $W(c)$ is dominated by possible interferences from other tasks. Furthermore, these interferences can be experienced by any client that is using the server. The main objective thus is to reduce the pessimism that is introduced through these interferences.

Less pessimistic results can be obtained by employing knowledge about the applications' behaviour and performing a schedulability and response time analysis for non-preemptive task systems (George et al. 1996). Enforcing a stronger coordination between tasks and server can help to further improve the analysis.

### 5.8.3. Single-Task Nodes

In a very simple system, one would execute each application thread on a separate MOSSCA node. The WCET of a thread then is composed of the times of sequential code execution and the waiting times that the thread experiences due to communication with other threads or servers. Methods to analyse such threads already exist, e.g. in the form of Hoare's communicating sequential processes (Hoare 1978). However, deriving WCWTs for communication points may still pose a challenge. Like in the case of MOSSCA servers, a stronger coordination between threads may ease the analysis and improve its results.

### 5.8.4. Local Multitasking

Executing only a single thread on each node is not desirable for real systems, as it would strongly restrict the flexibility of a system. Instead, node-local multitasking must be enabled. Regardless which multitasking approach is chosen, the results of the previous section 5.8.3 still apply, but must additionally heed execution delays that can be introduced through other threads.

Applying user-level scheduling results in a very low overhead for the OS, and timing analysability depends mainly on its application-level implementation. Similar considerations apply for cooperative task scheduling, where the application developers bear the main responsibility for analysability and correct behaviour.

A full scheduler can also be integrated in the local MOSSCA kernel to support classic real-time scheduling schemes (C. L. Liu and Layland 1973) and derivatives. Consequently, the well-known approaches for schedulability and response time analysis can be applied. Again, it may be possible to improve these results by enforcing a stronger coordination between tasks. Furthermore, it is important to keep in mind that even in safety-critical domains still applications may exist that have relaxed real-time constraints. By applying more specific scheduling approaches, system utilisation through these applications may be reduced and might be available to execute further, less critical applications that e.g. only have best-effort requirements. To exploit such behaviour, it is important to capture the existing tolerances in appropriate models from which formal execution parameters can be derived.

### 5.8.5. Coordination

So far, the works on MOSSCA assume that it is sufficient to provide bounded execution times for OS and application services. In this simple model, two problems can be identified: (1) Accesses to shared resources can happen any time, and special provisions must be taken to ensure predictability. (2) When applications or whole systems are ported to different platforms, execution patterns will change. Thus, accesses to shared resources will possibly have to be analysed again on a very fine-granular level. Using more specialised execution models can help to ease the analysis of shared resources and increase the portability of applications. Therefore, the OS needs to provide appropriate support.

### 5.8.6. Error and Shutdown

When the need for error handling arises, proper operation of the system may be interrupted at least partially for some time. Therefore, it is also necessary that such interruptions are considered during application development, and that the duration of any error handling routine can be bounded. OS routines used during error handling must also be analysable and predictable. If an error finally leads to a restart of the system, it is again important to also provide safe upper bounds for the duration of the boot process. During system shutdown, the OS plays only a minor role. Again, its mechanisms used during this phase must support the analysis. However, the main problems must be solved on application level, e.g. to bring the whole system into a safe state.

### 5.8.7. Timing Behaviour

As mentioned already in the above sections, a more comprehensive knowledge about the behaviour of applications might be exploited to achieve a more predictable behaviour of the whole system. Therefore, suitable tools are required to gather relevant properties of application and exploit these for the design and analysis of the system.

## 5.9. Summary

The MOSSCA architecture and implementation concepts provide the base for an OS for SCSs using manycore processors. MOSSCA is inspired by the factored operating system (see sect. 4.1.3). It is based on the concept of distributing application and OS functionality as far

as possible to exploit the large numbers of cores that will be available in future manycore processors. Compared to fos, it additionally responds to the requirements of SCSs. Although the design of MOSSCA assumes a rather restricted processor architecture, the concepts can also be applied to more general manycore processors, if appropriate mechanisms e.g. for handling components with low predictability are implemented. A number of use-case implementations shows, how the MOSSCA concepts can be be applied to for specific software (AUTOSAR) or hardware architectures (parMERASA and T-Crest platforms). The last section highlights several aspects that should be investigated to support the predictability and analysability of a MOSSCA system. While there exist simple solutions for most aspects, more sophisticated approaches can be expected to yield a better performance. In the next chapters, some of the mentioned aspects are investigated in more detail.

# 6 Predictable Boot Process

*This chapter is based on Kluge, Gerdes, and Ungerer 2014b*

To date, the boot process in multi-/manycore processors has received only minor attention in research. Research works about real-time software are typically concerned about the regular operation of the system, but do not care about bringing the system into an operational state. Concerning current processor architectures used in SCS, this poses only a minor problem. In single-core processors, bootstrapping can be performed in a strictly sequential manner which is analysable with current WCET analysis techniques. Boot-strapping in today's multi- and manycore processors is usually performed in the following manner (see e.g. Peters 2008; Intel 2013): one dedicated *coordination* or *bootstrap core* is the first to start executing code, while the other cores are in a wait state. The coordination core performs basic initialisations of the other core and then starts these. In the case of the Parallella manycore (Adapteva 2013c), at least the first core of the processor must be started by the external main processor. Then, a similar sequential approach can be applied. So again, a sequential approach is supported which does not pose any problems for WCET analysis.

However, as core numbers increase, bootstrapping the whole processor can be challenging concerning the total duration of the boot process. Data and code that should reside in the core's local memories must possibly be loaded from off-chip ROM, thus introducing long latencies. Even though a sequential approach can yield bounded WCET, this bound may be too high for certain applications. It is important to note that bootstrapping does not only occur when a system is started. Many systems employ watchdog timers or similar mechanisms for detecting and handling runtime faults. Often, when a fault is detected, the system is simply restarted from scratch even during regular operation. In such circumstances, it is very important that the restart is finished not only within a predictable, but ideally also short time span, thus allowing to bound blackout times. The parallelism available in manycore processors can be used to speed up the process of bringing the system into an operational state.

In this chapter, three approaches for bootstrapping a manycore-based system are investigated in terms of their WCD. Two state-of-the-art bootstrapping approaches and an optimised approach exploiting the parallelism in manycore processors are presented in section 6.1. The evaluation and analysis methodology is introduced in section 6.2. Results are presented in section 6.3. Section 6.4 identifies some directions into which this work can be continued. Section 6.5 sums up the important results of this chapter.

## 6.1. Bootstrapping

The bootstrapping approaches presented in this chapter are intended to be employed on a message-passing manycore processor with distributed memories and real-time capable interconnect (see sect. 5.1). Although they are based on the MOSSCA architecture, they may be applied to any system architecture that can fulfil the basic requirements stated in the descriptions of the approaches.

### 6.1.1. Preliminaries

An *operational system* shall be organised as follows: on each core, a small operating system kernel provides basic management of the node-local hardware unit. The kernel is the same on each node. Node-specific kernel configuration is achieved by separate configuration data. On top of the kernel, applications are executed.

The aim of the boot process is to provide any node in the manycore processor with code and data. This means that images containing kernel and application must be loaded from external ROM to the respective nodes. Bootstrapping is finished when each node has received its image(s) and is ready to execute its application. For generality, it is assumed further that each node executes a different application. This scenario represents the worst case that can happen at all. Having multiple nodes executing the same application code might only lower the time that is required for bootstrapping, if at all.

A dedicated node of the manycore processor shall be responsible for performing and coordinating bootstrapping of the whole processor. This node is denoted as the *bootcore*. The terms *nodes* or *each node* of manycore processor shall denote *all nodes except for the bootcore* in the following sections. The bootcore executes a special application *bootrom* that manages the bootstrapping of the whole processor. The application uses a *system configuration table* that indicates which core should execute which application.

### 6.1.2. Baseline: Full Image

The baseline is provided by the sequential full image (FI) approach. Here, kernel and application are linked into one binary image. The image also contains all configuration data. For each node, a different, single image must be loaded from ROM and sent over the NoC to the node. Then, the node can start to execute the kernel that sets up the local hardware units and finally starts the application.

### 6.1.3. Optimisation 1: Splitting Images

Assuming that the kernel is the same for any node, the FI approach performs redundant work by loading the kernel part multiple times from ROM. In the split image (SI) approach, a node's binary image is split into kernel and application images and node-specific configuration data for the kernel. The bootcore loads the kernel image only once from ROM and then sends it to all nodes consecutively. Thus, multiple long-latency loads from ROM performing each the same work are avoided. As soon as the kernel on the nodes has finished a basic initialisation, it notifies the bootcore. The bootcore then loads configuration data and application images separately from ROM and sends these to the respective node. When both chunks off data have been received, the local kernel can perform final configuration settings for its node and then is ready to start the application.

### 6.1.4. Optimisation 2: Self-Distributing Kernel

Using a SD relieves the bootcore from having to perform the distribution of the kernel image all by itself. Instead, the bootcore sends the kernel image to only few nodes of the processor. The kernel is extended by a mechanism that enables it to distribute itself to further nodes. The rules describing the pattern of this distribution are part of the kernel image. While the kernel distribution is in progress, the bootcore can already load configuration data and application images for nodes where the kernel has been started and finished self-distribution. This approach makes use of the parallelism that is available in the manycore processor.

## 6.2. Evaluation

A WCD analysis was performed for all three bootstrapping approaches. This section describes the tools and methodology that were used in the analyses, as well as the actual evaluation scenario.

### 6.2.1. Methodology

The bootstrapping approaches *Split Images* (SI, sect. 6.1.3) and *Self-Distributing Kernel* (SD, sect. 6.1.4) are integrated in the MOSSCA reference implementation (see sect. 5.4). From these implementations the sizes of images and configuration data as well as WCETs of sequential code can be derived. The operativeness of both the SI and SD implementations are validated on the MacSim manycore simulator (Metzlaff et al. 2011). The runtimes and sizes for the *Full Image* (FI, sect. 6.1.2) approach are calculated from an adjusted implementation of SI, assuming that kernel and application reside within one image. WCETs of sequential code are obtained using the OTAWA WCET analysis tool (Ballabriga et al. 2010) assuming execution on a ARMv7 processor. These WCETs do not include the time that is needed for loading data from ROM and for sending/receiving data via the NoC. This information is added in the next step.

The worst-case duration of the full boot process is calculated through an abstract simulation of the actual worst case execution path. Therefore the *Manycore WCET Simulation (`mwsim`)* tool was developed, which is described in section 6.2.2. In these simulations all other timing parameters that do not stem from sequential execution are determined.

### 6.2.2. `mwsim`

The `mwsim` tool is used to calculate the actual WCD of bootstrapping the whole manycore processor. The input of `mwsim` is divided in three parts: (1) a system description contains relevant parameters of the underlying hardware (e.g. number of cores, memory and NoC latencies); (2) global variables for fine-grained configuration of (3) script files that contain an abstract representation of execution steps that are performed by the nodes of the manycore processor. System parameters are used to determine the execution times of certain execution steps specified in the script file. The relevant parameters are mentioned in the description of script commands below. Global variables are used to achieve a greater flexibility in writing script files.

Table 6.1.: Overview of `mwsim` script commands

| *Command* | *Description* |
|---|---|
| `exec` *c* | sequential execution for *c* cycles |
| `load` *n* | load *n* bytes from ROM (word-wise) |
| `send` *dst id n* | send a message named *id* with *n* bytes to node *dst* |
| `wait` *id* from *src* | wait for a message named *id* from node *src* |
| `recv` *src id* | receive a message named *id* from node *src* |
| `parwait` *id* `from` *lst* `as` *src* | wait for a message named *id* sent by the nodes specified in list *lst*, perform specified subprogram for each message |
| `foreach` *val* `in` *lst* | iterate over list, store current value in variable *val* and execute specified subprogram |
| `calc` *vt* = *ex* [+-*/] *ex* | perform basic arithmetic calculation and store result in variable *vt* |

### Script Commands

An overview of the script commands with their syntax is provided in table 6.1. The following paragraphs describe the commands, their functionality and how they affect the overall WCD in detail. Some commands rely on a parameter `wordsize` which must identical throughout the whole system. They also may use other system parameters which are shown in table 6.2.

**Sequential Execution** The `exec` command represents a sequential execution of code that uses only local resources. A parameter *c* specifies the WCET of the corresponding code block. *c* is derived with the OTAWA tool.

**Loading from ROM** `mwsim` supports loading from only a single type of ROM. It is modeled as a flash memory and characterised by the following system parameters: Reading one word from ROM incurs a `load_memory_latency`. Transferring the load request resp. the data over a bus connection adds a latency of `load_bus_latency` in total for each word. Usually large blocks of data are loaded from ROM, which is performed inside a loop. This behaviour is modeled by the `load` command through additional parameters. The loop has a fixed `load_init_latency` that is incurred once. An additional `load_exec_latency` is incurred in every loop cycle when one word is loaded. Both parameters must be determined with the WCET analysis tool. Load operations shall only be executed by the bootcore.

**Sending over NoC** The `send` command models the sending of a message to another node. To achieve an easy assignment to corresponding `wait`/`recv` commands, each message has an ID. The ID needs not to be unique, as messages are additionally distinguished by their sender/receiver and timestamps. In the case of the `parwait` command (see below) the ID must even be the same for all sender nodes. Within a message, one or multiple words are transmitted. Similar to the `load` command, the actual implementation performs this in a loop having a `send_init_latency` and an additional `send_init_latency` for each data word. Both values must be derived through WCET analysis. They represent only the latencies that the sending node experiences. Sent messages are put in a simulator-internal buffer with size `tx_buffer_size` for delivery at the appropriate time. If the buffer is contended, the sender

Table 6.2.: Overview of `mwsim` system parameters

| *System Parameter* | *Unit* | *Description* |
|---|---|---|
| `wordsize` | Bytes | System-wide size of words |
| `load_memory_latency` | Cycles | |
| `load_bus_latency` | Cycles | |
| `load_init_latency` | Cycles | |
| `load_exec_latency` | Cycles | |
| `send_init_latency` | Cycles | |
| `send_exec_latency` | Cycles | |
| `recv_init_latency` | Cycles | |
| `recv_exec_latency` | Cycles | |
| `tx_buffer_size` | Bytes | Size of a node's receive buffer |
| `tx_period` | Cycles | Minimum time distance between two messages from one sender to the same receiver |
| `tx_wctt` | Cycles | Worst-case traversal time of a single message |

experiences additional latencies. These depend on the rate the NoC transmits the messages, in `mwsim` modeled by the `tx_period` parameter. The process of message delivery will be more detailed in sect. 6.2.2.

**Waiting for NoC Message** Before a message can be received, the program running on the node has to ensure that the message has actually arrived yet. This is achieved by the `wait` command that allows a node to wait for a message identified by a certain ID and a sender node. The duration of the `wait` command depends on the time the relevant message is actually sent and its transmission time over the NoC. `wait` returns when the first word of a message arrives.

**Receiving a Message from NoC** When a message arrives at a node, the program has to explicitly receive the data using the `receive` command. Actual receiving off the message, i.e. moving data from the NoC receive buffer into memory is performed in a loop with parameters `recv_init_latency` and `recv_exec_latency`. If a transmission is still in progress during execution of the loop, the loop latency may be increased to the period with which the words of the message arrive.

**Waiting for multiple nodes** In certain circumstances, a node is waiting to receive messages of the same type from several nodes. `mwsim` supports such behaviour through the `parwait` command. The command specifies a message name and a list of nodes that will send this message. When the specified message arrives from one of the nodes, a subprogram is executed. If the specified message has been received from all nodes in the list, program execution continues. The WCD of this loop consists of the waiting times for all messages and the corresponding executions of the subprogram. Execution times for the loop's actual implementation must be added using `exec` commands.

**Loop** `mwsim` provides a `foreach` loop that allows to iterate over a list of values and execute the same subprogram for each. No execution time is spent for this command itself, loop overheads in the actual implementation must be added using `exec` commands.

**Elementary calculations** Finally, `mwsim` provides means to perform basic arithmetic calculations. These are intended to ease writing of the script files. Execution of the `calc` command consumes no processing time. Operands of the calculations can be either variables or numbers. Calculation results are stored in variables.

### Message Transmission

Any time a node executes a `send` command, a message is generated and the worst-case send and arrival times of the first and the last words are calculated. `mwsim` models a NoC with predictable worst-case traversal times (WCTTs). Message words from the same sender to the same receiver can be sent every `tx_period` cycles over the NoC. They are transmitted with a WCTT `tx_wctt`. The first word must wait at most `tx_period`-1 cycles for being sent, transmission takes another `tx_wctt` cycles. After the first word, each `tx_period` another word is transmitted.

### Script Execution

Algorithm 6.1 shows how `mwsim` executes scripts. First, each node is warmed up (lines 1-3). `mwsim` executes a node's script until a waiting point (`wait` or `parwait` command) is reached. During this phase, messages are buffered internally, but not yet delivered. When all nodes have reached a waiting point, buffered messages are used to control further execution. This node-by-node execution phase (lines 4-7) works as follows: The message with the smallest arrival time of the first word is removed from the buffer and delivered to the receiving node where it is put into another buffer. If the node has been waiting for the message, it continues execution until it hits a waiting point for a message that was not yet delivered. Otherwise, if the node is not yet waiting for the delivered message, it continues to wait, and another message is delivered.

Simulation finishes under two conditions: (1) if all nodes are in a waiting state and no more messages are queued for delivery, or (2) if script execution is finished on all nodes. Condition (1) should usually not happen and indicates an error in the script files. Condition

**Algorithm 6.1:** Script execution in `mwsim`

```
   // Warm up nodes
1  foreach node n do
2  |  execute n until waiting;
3  end
   // Node-by-node simulation
4  while not all nodes finished ∧ messages available do
5  |  deliver message with smallest arrival time;
6  |  execute receiver until waiting;
7  end
```

(2) represents the case, where all work that was originally defined also was performed by the nodes. Then, `mwsim` yields the execution times $c_i$ (including waiting times) for each node $i$. The maximum of these values represents the WCD bound of the whole boot process $c_B$:

$$c_B = \max_i c_i \tag{6.1}$$

The results obtained through `mwsim` do not allow to assign a WCD smaller than $c_B$ to a single node. Simulation with `mwsim` assumes that any process in the system happens with its worst-case timing. Actual execution might diverge from this worst-case behaviour. This can result in messages arriving in a order different from the one simulated by `mwsim`. Especially, the request for some node's image may arrive earlier at the boot core than derived by `mwsim`. If this request is also processed earlier by the boot core, other requests will be processed later than in the worst-case found by `mwsim`. This will result in a reordering of the places of the time intervals during which the respective requests are served by the boot core. Some nodes may finish their startup process in a different order. The influence of this behaviour on the total duration of the boot process is discussed in more detail in sect. 6.2.4 after the actual boot process scenario has been introduced.

### 6.2.3. Scenario

The general system parameters used in the evaluations are shown in table 6.3. The parameters for ROM access are obtained assuming a NOR flash memory with a read latency of 80 ns (Crippa et al. 2008) and a clock frequency of 400 MHz. The initialisation and execution latencies of the load, send, and receive loops are obtained from the implementation using OTAWA. Parameters concerning the NoC are chosen based on the actual size of the NoC that is used. The assumed NoC has a latency of one cycle per hop. WCTTs (`tx_wctt`) assume the maximum possible hop count of the respective NoC. The definition of `tx_period` and are based on the bounds that can be found in (Schoeberl et al. 2012). All values are shown in table 6.4.

Table 6.3.: Global system parameter settings for evaluations

| *System Parameter* | *Value* | *Unit* |
|---|---|---|
| wordsize | 4 | Bytes |
| load_memory_latency | 32 | Cycles |
| load_bus_latency | 2 | Cycles |
| load_init_latency | 76 | Cycles |
| load_exec_latency | 13 | Cycles |
| send_init_latency | 76 | Cycles |
| send_exec_latency | 13 | Cycles |
| recv_init_latency | 76 | Cycles |
| recv_exec_latency | 13 | Cycles |
| tx_buffer_size | 512 | Bytes |

The implementation of the three bootstrap approaches assumes that no errors occur during execution. If messages consisting of more than one word must be transmitted, they are

Table 6.4.: System-size-specific parameter settings for evaluations; all values in clock cycles

| *#Cores* | 4 | 9 | 16 | 25 | 36 | 49 | 64 | 256 | 1024 |
|---|---|---|---|---|---|---|---|---|---|
| tx_period | 3 | 8 | 16 | 32 | 54 | 86 | 128 | 1024 | 8192 |
| tx_wctt | 2 | 4 | 12 | 8 | 10 | 12 | 14 | 30 | 62 |

preceded by a message containing the actual size of the data. The implementation does not use message names. However, a correct receive order is ensured by the assumed fault-free execution.

In all evaluations, node 0 acts as the dedicated bootcore. Kernel self-distribution for the SD approach is implemented in the following manner: The bootcore (B) is located at lower left corner of the mesh NoC (see figure 6.1). It sends the kernel image to its row- and column-wise neighbours. Nodes in the first column distribute the image first to the next row (if existing), and then to their neighbour in the same row. Nodes from the second to the penultimate column perform only row-wise distribution. Nodes in the last column perform no further distribution.

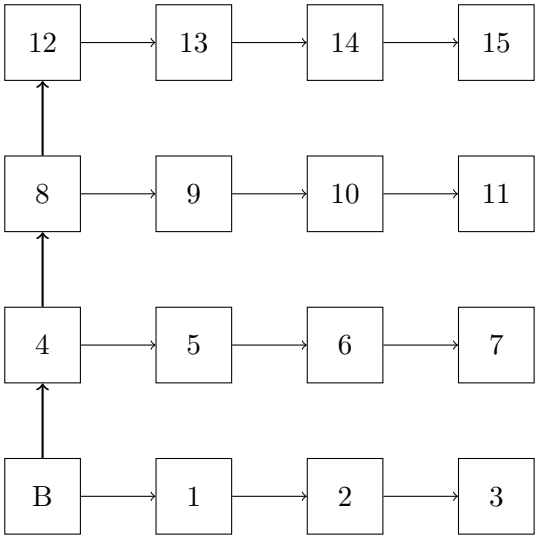



Figure 6.1.: Kernel distribution scheme; B = bootcore (core 0)

Exemplary scripts for the SD approach running on a 64-core processor are shown in algorithms 6.2 and 6.3. The following paragraphs describe these scripts and point out differences to the FI and SI bootstrapping approaches.

On the bootcore (alg. 6.2), first a special kernel is loaded by a minimal bios (line 3). The kernel itself loads and starts the bootrom application (lines 5-6), which performs the actual task of bootstrapping. First, it loads the kernel that shall run on all other nodes and distributes it to two nodes (lines 8-11). In the SI approach, the kernel is sent directly to all other nodes at this point. In the FI approach, this part is completely left out. Then, the system configuration table is loaded. Inside the `parwait` loop (lines 15-21), the bootrom application waits for a *KRDY* message from any other node, which is sent as soon as the local kernel has started. Upon arrival of such a message, the bootrom application loads the node's

**Algorithm 6.2:** Script file for the bootcore in the SD approach

```
1  node [0]
2      exec 136
3      load SIZE_KERN0
4      exec 359
       # load bootrom application
5      load SIZE_BOOTROM
       # call application
6      exec 15
       # bootrom
7      exec 81
       # load kernstd
8      load SIZE_KERNSTD
9      exec 284
       # start distribution
10     send 1 KSTD SIZE_KERNSTD
11     send 8 KSTD SIZE_KERNSTD
       # configure system
12     exec 1008
13     load SIZE_CFG_SYS
       # check & perform configuration
14     exec 110
15     parwait KRDY from [1-63] as cnode
           # configure node
16         exec 487
           # first load configuration data
17         load SIZE_CFG_NODE
18         send cnode CONFIG SIZE_CFG_NODE
           # ...then application code
19         load SIZE_APPLICATION
20         send cnode APPLICATION SIZE_APPLICATION
21         exec 222
```

configuration data (lines 17/18) and application image (lines 19/20) and sends it to the node. There is no further synchronisation between the transmission of these both packets. The receiver loads both of them first into its memory before it finishes the node's configuration.

**Algorithm 6.3:** Script file for nodes with column and row distribution in the SC approach

```
1  node [8,16,24,32,40,48]
       # bios
2      exec 128
3      wait KSTD from dist
4      recv dist KSTD
       # kernel
5      exec 442
       # distribute kernel
6      calc ndestc = MY_NODE_ID + 8
7      send ndestc KSTD SIZE_KERNSTD
8      calc ndestr = MY_NODE_ID + 1
9      send ndestr KSTD SIZE_KERNSTD
       # kernel ready
10     send 0 KRDY 1
11     wait CONFIG
12     recv 0 CONFIG
13     wait APPLICATION
14     recv 0 APPLICATION
       # apply configuration
15     exec 785
       # start application
16     exec 130
```

Algorithm 6.3 shows the script for a regular SD node that performs column- and row-wise distribution of the kernel. Similar to the boot node, waiting for and loading the kernel is modeled by a small bios. The kernel then is distributed in lines 6-9. In line 10, the *KRDY* message is sent to the bootcore. In both the FI and SI approaches, no kernel distribution is performed at this point. Then, the node waits for configuration data (lines 11/12) and application image (lines 13/14) and finally prepares the node for application execution (lines 15/16). While the kernel image is the same for all nodes except the bootcore, several scripts are used to model the SD approach. The scripts only differ in the number of re-distributions of the kernel, where the column-wise or both `send` commands (lines 7 and 9) are left out. The execution times of the kernel in line 5 are adjusted accordingly by providing appropriate flow facts to OTAWA. The flow facts ensure that the analysis assumes a certain path in which no or only one distribution of the kernel is performed.

Table 6.5 shows the sizes of kernel and application images, and configuration data used in the evaluations. In the SI and SD approaches, the application images are the same. The SI standard kernel is slightly smaller as it does not contain the self-distribution mechanism. The boot kernel is the same in both approaches, as its task is always to load the bootrom application that performs the actual coordination of the boot process.

Table 6.5.: Sizes of kernel and application images, in bytes

| *Full Image (FI)* | |
|---|---|
| Boot image | 8,572 |
| Full image | 14,668 |
| *Split Images (SI)* | |
| Boot kernel | 4,452 |
| Bootrom application | 4,208 |
| Standard kernel | 4,460 |
| Application | 10,240 |
| *Self-distributing Kernel (SD)* | |
| Boot kernel | 4,452 |
| Bootrom application | 4,304 |
| Standard kernel | 4,792 |
| Application | 10,240 |
| *Configuration data* | |
| System config (per node) | 180 |
| Node config (only SI, SD) | 512 |

### 6.2.4. Correctness

The WCD bound derived through `mwsim` for the above scenario needs some closer inspection. Especially, it is important to clear whether $c_B$ from eq. (6.1) can represent a *safe* upper bound for the boot process. Therefore it is important to note that in the scenarios presented above the boot process takes place in non-interleaving phases that are coordinated through the bootcore. If the bootcore distributes the kernel to all cores (SI approach), this task is completed before any request for configuration data and application images is handled. Insofar, the `parwait` command performed by the bootcore can be seen as a kind of a barrier. Once the bootcore has entered this loop, it will perform no other tasks until it has served all client nodes. For a simple case where all nodes execute the same code, the following lemma can be derived:

**Lemma 6.1.** *Consider two client cores that are served subsequently in a schedule generated by `mwsim`. Assume that both cores receive the same amount of data from the bootcore. Then, changing the execution order of these two cores does not change $c_B$ in eq.* (6.1)*.*

*Proof.* Let the requests $R_1, R_2$ for the client nodes $n_1, n_2$ arrive at times $a_1, a_2$ with $a_1 < a_2$ in the schedule generated by `mwsim`. Further, let $h_1, h_2$ be the times when the bootcore has finished handling the corresponding requests, and $r_1, r_2$ with $r_1 < r_2$ be the times when the corresponding responses have fully arrived at the client nodes. $R_2$ can experience a delay through $R_1$, if $a_2 < h_1$. The delay can be calculated as $d = \max\{h_1 - a_2, 0\}$.

Now consider the same schedule, but with $R_1, R_2$ arriving in reverse order at times $a'_2 = a_1$ and $a'_1 = a_2$. As the system is in the same state at $a'_2 = a_1$ as in the original schedule and $R_2$ has the same complexity as $R_1$, $R_2$ will be finished at time $h'_2 = h_1$. With the same argument, serving $R_1$ then is finished at time $h'_1 = h_2$. Consequently, the responses arrive at times $r'_2 = r_1$ and $r'_1 = r_2$ at their originating cores. □

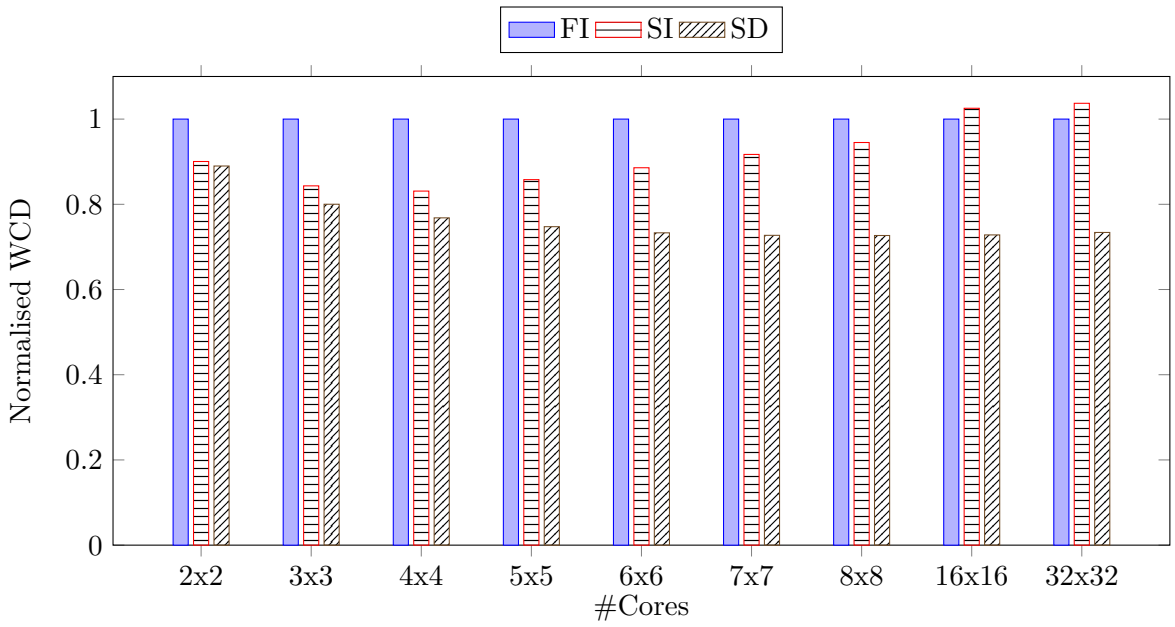



Figure 6.2.: WCDs of the boot process for different processor sizes; normalised to the duration of FI

With the help of lemma 6.1, the following theorem can be derived:

**Theorem 6.1.** *Assuming the preconditions from lemma 6.1 for all client nodes, the duration of the boot process is safely bounded by $c_B$ from eq.* (6.1) *regardless in which order the requests actually arrive during execution.*

*Proof.* The request order of an actual execution can be obtained from the order found by `mwsim` through multiply swapping arbitrary subsequent requests (in the manner of the bubblesort algorithm). As each single swapping cannot impact the total duration, $c_B$ is preserved as a safe upper bound. □

Theorem 6.1 ensures for a specific scenario that $c_B$ from eq. (6.1) is actually a safe upper bound. Unfortunately, it may not apply for the general case where application images have different sizes. In this case, a different execution order may result in an actually higher WCD of the boot process than derived by `mwsim`. To get still a valid WCD with `mwsim`, one has to overestimate the WCD by assuming the maximally occurring application image size for all nodes.

## 6.3. Results

Figure 6.2 compares the WCD bounds $c_B$ (see eq. (6.1)) of the three bootstrapping approaches on different processor sizes. The bars indicate the time the last core is ready to execute its application ($c_B$ from eq. (6.1)). All values are normalised to the FI approach for each number of cores. The underlying absolute numbers can be found in table 6.6. For the core numbers that are examined, the SD approach has an advantage of 11-27% over the FI approach. Also,

it performs in all cases better than the SI approach. This advantage stems from the fact that the self-distribution mechanism of the kernel in the SD approach relieves the bootcore strongly from the task of kernel distribution. In FI, the bootcore has to load the kernel that is contained in each image separately for each other core. In the SI approach loading the kernel happens only once. However, both the FI and SI approaches suffer from contention of the send buffer of the bootcore (see below). Contrary, in the SD approach the bootcore can much earlier start loading applications. Furthermore, up to 8x8 cores, the SI approach performs better than the FI approach. However, when the total number of cores on a processor increases, SI is outperformed by the FI approach as shown in figure 6.2 for 16x16 and 32x32 cores. This behaviour is mainly caused by the `tx_period` parameters. Due to the high value of this parameter in the 16x16 and 32x32 scenarios (see table 6.4), transfer of data from the boot core's send buffer is about two orders of magnitude slower than the transfer from the core's memory to the buffer (`send_exec_latency`, table 6.3). Thus, the boot core is slowed down while it is sending the kernel to all other nodes. While this behaviour affects also the FI approach, the SI approach has an additional synchronisation point where the boot core waits for *KRDY* messages from the other cores (similar to line 15 in algorithm 6.2). After this synchronisation point, the boot core is again sending data and therefore waiting most of the time for the data being transmitted from the send buffer. Further simulations show that if `tx_period` is decreased, SI can perform better than the FI approach also for large systems.

Table 6.6.: WCDs of the boot process

| Cores | FI | SI | SD |
|---|---|---|---|
| 2x2 | 770,900 | 694,171 | 686,010 |
| 3x3 | 1,884,712 | 1,589,378 | 1,508,050 |
| 4x4 | 3,580,181 | 2,975,200 | 2,750,241 |
| 5x5 | 7,024,712 | 6,026,258 | 5,250,923 |
| 6x6 | 12,921,407 | 11,445,768 | 9,472,063 |
| 7x7 | 23,118,166 | 21,200,584 | 16,810,684 |
| 8x8 | 39,674,211 | 37,493,774 | 28,834,226 |
| 16x16 | 968,858,078 | 993,482,216 | 705,480,325 |
| 32x32 | 30,732,373,725 | 31,871,846,984 | 22,554,961,381 |

Figure 6.3 gives a detailed view of the finishing times of the single cores in a 8x8-core system as simulated by `mwsim`. Execution times in the FI and SI approaches increase linearly with numerical distance from the bootcore. For numerical lower cores, the FI approach yields a better performance than the SI approach. This is due to the fact that the cores in SI, before receiving their application, must wait for the bootcore until it has sent the kernel to all cores. For low core numbers, SI can compensate for this behaviour in cores that are started later, as it incurs the overhead of loading the kernel only once. Concerning the SD approach, a special behaviour that is introduced through the self-distributing kernel can be observed: The kernel is distributed preferably in the first column, and just then into the current row (see fig. 6.1 and alg. 6.3). Nodes nearer to the first column get ready earlier for receiving their application than those at the end of the row. Even if a node resides in a higher row, it may be ready earlier than a node at the end of a lower row. The local minima in the durations are experienced by cores residing in the first column. In some scenarios, cores residing at the end

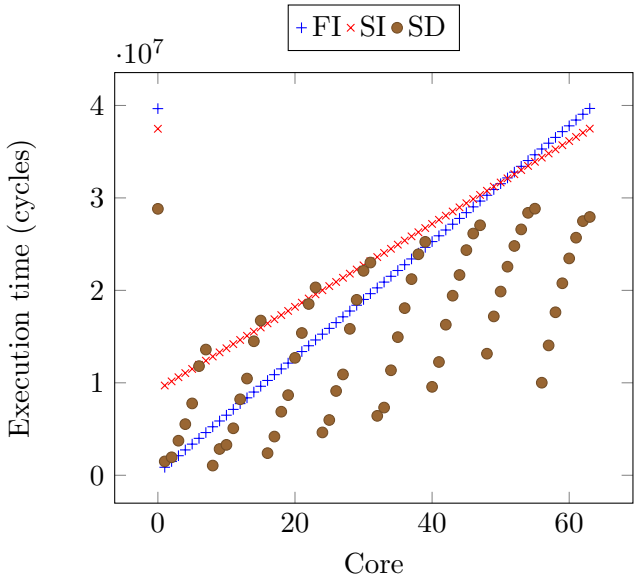



Figure 6.3.: Durations experienced by the single cores in a 8x8 manycore during bootstrapping

of the first few rows may actually be ready later than their counterpart in the SI scenario. This stems from the fact that their kernel arrives slightly later than in the SI scenario due to the distribution mechanism. However, only few cores are affected by this minimum worsening. Still, the maximum startup time using SD is clearly lower than with the SI or FI approach. Another particularity of the SD approach in processors with 5x5 and more cores can also be seen in figure 6.3: In the `mwsim` simulation, the first core in the last row becomes ready earlier than the first core in the penultimate row (cores with the lowest execution times in the last to segments in fig. 6.3). This deviating behaviour stems from the fact, that the first core in the last row has to perform only one kernel distribution. Due to the short startup execution times of the kernel, the node thus can send its *KRDY* message to the bootcore much earlier. Please note that the actual execution behaviour may look different from the one found by `mwsim`. Especially the order in which the single cores get ready may differ depending on the arrival order of *KRDY* messages at the bootcore. However, the WCD bound for the whole boot process is guaranteed as long as the conditions for theorem 6.1 are fulfiled.

Finally, the absolute values of the WCDs found in the different scenarios need some discussion. In addition to the total runtimes, `mwsim` also logs the times that a node spends waiting for a message. In the scenarios up to 6x6 cores, the boot core spends actually no time waiting for any messages, meaning that these already arrived when the program hits a `wait` instruction. In the 7x7 and larger scenarios using the SD approach, the wait times on the bootcore amount to less than 0.1% of the WCD. In the FI and SI approaches, the wait times of the bootcore are 0 also for the larger scenarios. The critical part of execution is the loop that distributes configuration data and application images to the cores. Both data and images must be loaded from slow ROM, and then be sent over the NoC. Most critical appears the NoC send buffer and the `tx_period` with which the buffer is emptied. As long as the buffer is full, the node cannot send further data and must wait until at least one word was actually injected into the NoC. If chunks of data are sent that are large compared to the

NoC buffer size, the node spends also a large amount of time in waiting for free space in this buffer.

Enlarging the buffer size until at least one of the data chunks fits completely into the buffer achieves only minor improvements. Programs sending data can put data at a very high rate into the buffer, limited only by their send loop latency of 13 cycles (see tab. 6.3). For cores numbers beyond 3x3, the `tx_period` clearly is the limiting factor (see tab. 6.4). Due to the relatively low rate with which data is moved from the buffer to the NoC, the buffer is saturated very fast such that subsequent calls of the `send` instruction still have to wait for the NoC. For processors with 2x2 or 3x3 cores, the loop latency is the limiting factor in the presented scenario.

## 6.4. Potentials for Further Work

The techniques implemented in `mwsim` provide a baseline for the analysis of bootstrapping and other processes. They may be extended in several directions.

### 6.4.1. DMA Units

Simulation in `mwsim` assumes that all communication between nodes is based on messages that are explicitly sent over resp. received from the NoC. In contrast, today's processors usually provide direct memory access (DMA) units to relieve the cores from memory-to-memory copy operations. Real-world applications exploit such DMA units very often, e.g. when running on multiple cores using a single memory. Recent manycore processors like the Parallella computer also possess DMA units. Such units are not yet modeled in `mwsim`. It is to be expected that using DMA will provide a certain speedup as they can relieve the boot core from some pressure.

### 6.4.2. Use of Best-Effort NoC

The results presented in the previous section 6.3 show that the waiting times the bootcore experiences during sending large chunks of data amount for a large part of the actual WCDs. This behaviour is part of the price to be payed for deterministic communication. As long as only small amounts of data are transmitted, this price is tolerable. In the bootstrapping scenario where large chunks of data must be moved to all cores, this price seems quite high. In contrast, best-effort NoCs may provide higher throughput rates, but usually cannot give the guarantees required by hard real-time systems in general, as their behaviour in situations with high contention may become non-deterministic. However, in the special case of the boot process, NoC communication is quite restricted. In the FI and SI scenarios, large chunks of data are only sent by the bootcore, while the other cores respond with only very small messages, if at all. In the SD scenario, additional chunks of data are sent between some neighbouring cores during kernel distribution. There are no communication streams competing for the same NoC connection in any scenario. Thus, it might also be possible to use a best-effort routing approach during bootstrapping with perhaps only few constraints to achieve a speed-up, and still keep the predictability of the boot process.

To get an impression of the possible speed-up the evaluations from section 6.3 are repeated using an optimistic NoC scenario. It differs from the scenario defined in section 6.2 only in the value of `tx_period` which was set to 1, meaning a core can send a one-word message at

Table 6.7.: Comparison of WCDs for a real-time and an optimistic NoC, SD approach

| Cores | RT-NoC | Optimistic NoC | opt/rt |
|---|---|---|---|
| 2x2 | 686,010 | 686,010 | 1.000 |
| 3x3 | 1,508,050 | 1,508,050 | 1.000 |
| 4x4 | 2,750,241 | 2,658,906 | 0.967 |
| 5x5 | 5,250,923 | 4,138,578 | 0.788 |
| 6x6 | 9,472,063 | 5,947,066 | 0.628 |
| 7x7 | 16,810,684 | 8,084,370 | 0.481 |
| 8x8 | 28,834,226 | 10,550,490 | 0.366 |
| 16x16 | 705,480,325 | 42,116,826 | 0.060 |
| 32x32 | 22,554,961,381 | 168,382,170 | 0.007 |

any time. The actual sending rate of a core in this scenario thus is limited by the execution time of the loop that performs the actual sending. Resulting WCDs for the SD approach are shown in table 6.7 and compared to those for a real-time NoC. While there is no gain for small core numbers, bigger processors can gain speed-ups up to two orders of magnitude. However, this approach requires a NoC like Æthereal (K. Goossens et al. 2005) that supports both best effort and guaranteed service routing.

### 6.4.3. Online Reconfiguration

The techniques presented in this chapter are not restricted to an analysis of the boot process. They might also be used to analyse scenarios of partial or complete reconfiguration of the system. This requires that the target configurations of the reconfiguration scenarios are known beforehand. As the behaviour of an SCS during its life cycle must be analysed statically in any case, this requirement can be assumed to be fulfiled. However, at least during a partial reconfiguration, it may not be possible to exploit speed-ups enabled by best-effort NoC routing, as some nodes in the system are still operating regularly and thus collisions could not be excluded.

### 6.4.4. General Timing Analysis

A further possibility would be to use the techniques presented in this chapter for the WCET analysis of distributed parallelised applications that use message-based communication. An important issue in the analysis of parallelised applications is to derive WCWTs. This might require to extend the `mwsim` tool: The current implementation can only give a bound on the whole process (eq. (6.1)), while the numbers found for single nodes my deviate. It would be necessary to explore all possible orders in which the nodes may arrive at an synchronisation point.

## 6.5. Summary

This chapter investigates the real-time capability of the boot process in a manycore processor. Two state-of-the-art approaches (FI, SI) for provisioning all cores with their code and data are compared. A third approach using a self-distributing kernel is proposed. The SD approach

exploits the parallelism inherent to a manycore processor by using a self-distributing kernel, thus relieving the bootcore from some load. To evaluate the three approaches, the `mwsim` tool was developed. `mwsim` simulates the boot process based on scripts that model the actual processes to calculate a safe upper bound for the time when all cores have received their code and data, and are ready to execute their application. The evaluations show that the SD approach outperforms both the FI and SI approaches. It can reduce the WCD of the boot process by up to 27% compared to the FI approach. Concerning the state-of-the-art approaches, the evaluations show a trade-off point in the number of cores: For small core numbers, the SI approach yields the higher performance, while for high core numbers FI might perform better.

Several improvements are discussed that can be applied to this work: So far, DMA units have not been considered to relieve the boot core from the work of sending large chunks of data. Further experiments show how a best-effort NoC can improve the duration of boot-strapping while still being able to provide a safe WCD bound. The analysis techniques implemented in `mwsim` can also be used for the timing analysis of reconfiguration scenarios, and they may be extended for the general timing analysis of parallel applications using message communication.

The results obtained with `mwsim` can be exploited under at least two aspects. First, if the maximum image size is assumed, the WCD found by `mwsim` represents a safe bound for the boot process in an event-triggered system. Second, a schedule found by `mwsim` can be used for a time-triggered implementation of the boot process that ensures an process behaviour as simulated with `mwsim`. In the second case, actual image sizes can be used for `mwsim` script execution, as the time-triggered implementation ensures the specified execution order.

# 7 Modeling Timing Parameters of Cyber-Physical Systems

*This chapter is base on Kluge, Gerdes, Haas, et al. 2014*

The term cyber-physical system (CPS) conveys the close integration of physical processes with control computers (Lee 2008). Computations in CPSs are often feedback loops where physical processes and computations mutually affect each other. Interaction between both is mediated by sensors and actuators. Insofar, a SCS in the context of this work is always a CPS, too. Computations in CPS must heed the constraints imposed by the physical processes they control. They are reactive with respect to the physical processes and must adapt to the passage of time and the intrinsic concurrency in physical systems.

**Example 7.1.** *In the following sections, the open source engine management system FreeEMS (FreeEMS n.d.) will be used as an example to illustrate the GTM formalisms. More details about FreeEMS can be found in appendix B.3 and (Kluge and Ungerer 2015). Like any other engine management system (EMS), FreeEMS uses number of sensors to monitor the current state of the engine and its environment. Concerning the engine, sensor(s) mounted to the crank and/or cam shaft play a central role. In the following, a crank shaft sensor is assumed for simplicity. It is used to raise interrupts in the engine management unit (EMU) on certain angularities of the crank shaft. Depending on the actual angularity, the EMS sets timers to trigger fuel injection or ignition. In the calculation of the trigger times, the EMS takes the engine's current speed and other physical states into account. Still, computations must be performed with low latencies heeding the timing behaviour of the engine. Their activation frequency depends directly on the angular speed of the crank shaft and changes over time. In the short term, this behaviour may be strictly periodic, but in the long term the frequency undergoes changes due to varying engine speeds. In the following, such behaviour will be termed* pseudoperiodic*. Additionally, FreeEMS implements functions that are executed strictly periodically.*

To ensure correct behaviour of the EMU, a schedulability analysis must be performed. A simple analysis can be based on the periodic (C. L. Liu and Layland 1973) or sporadic (Mok 1983) task models. However, its results may be quite pessimistic, as it would have to assume the maximum possible activation frequency. Specialised task models like elastic (G. C. Buttazzo et al. 2002) or rate-adaptive (G. C. Buttazzo et al. 2014) tasks are able to map the behaviour of such computations more precisely and thus introduce less pessimism into the analysis. Still, they discard some knowledge: While they can handle changes in activation

frequency of tasks, they do not map further physical constraints like bounds to the rate of frequency change that are inherent in the engine. The fact that the frequency cannot change abruptly, but only gradually, might be used for further optimisation of the whole system, e.g. by giving online guarantees about available processing time in the short term.

The aim of this chapter is to develop a modelling technique (the generic timing model, GTM (Kluge, Gerdes, Haas, et al. 2014)) that aims to capture timing parameters of CPSs in arbitrary granularity. These can be, e.g. timing constraints imposed by the physical world, or timing requirements that are derived from a requirement specification, but also tolerances in the timing behaviour that might be exploited. In the following section 7.1, related work is reviewed. Section 7.2 introduces the system model on which the GTM formalisms (sect. 7.3) are based. Periodic and pseudoperiodic behaviour is examined in more detail in section 7.4. A summary of this chapter's results can be found in section 7.5.

## 7.1. Capturing "Time" During System Design

To capture timing parameters of applications and exploit them in system design, various approaches have been developed. In OASIS (David et al. 1998), timing information is part of an application's source code and used by the OS to coordinate communication between tasks. The central target of the OASIS approach is to guarantee safety properties of a system. Some ideas of the OASIS approach can also be found in the time-triggered language Giotto (Henzinger et al. 2003), which allows to specify functionality and timing of embedded control systems. It introduces the concept of LET. With the LET concept applied, composability and portability of applications is eased. The *Timing Definition Language (TDL)* (Pree and Templ 2004) is based on the Giotto concepts, but introduces its own syntax and a module concept that allows to model multiple applications running on the same system. However, the Giotto concept is based on periodic tasks and thus is not easy to apply for cases with sporadic reactions with short latencies like an EMU. Both the OASIS and LET approaches have in common that they purport a time pattern that is ensured by execution platform. Application development must consider this pattern to benefit from the approach. As they are mainly based on periodic processes, they can hardly convey a behaviour such as the one found in the EMU use-case.

The following approaches leave greater flexibility, as they are centered more on the application with the aim to bring timing information into the execution platform. Espinoza et al. (Espinoza et al. 2006) present an approach to annotate UML MARTE models with non-functional properties. The *Timing Augmented Description Language (TADL)* (Blom et al. 2009) extends EAST-ADL2 (D. Chen et al. 2008) for the annotation of timing constraints. TADL V2 (Peraldi-Frati et al. 2012) provides additional support for multiple clock bases that need not necessarily be based on physical time. This approach is able to capture crank angle interrupts as a time base. It can also represent the relation between different time bases, even if the drift is variable. Though, the evolution of this drift cannot be constrained for the special case of pseudoperiodic events.

*Event streams* (Gresser 1993; Albers et al. 2006) are used to determine upper bounds for event arrival rates in event-triggered real-time systems and perform a schedulability analysis. The concept of event streams, among others, is picked up in the *real-time calculus* (Thiele et al. 2000; Chakraborty et al. 2003) for the analysis of real-time systems. Hanbo et al. (Hanbo et al. 2009) use the duration calculus (Hansen and Chaochen 1997) to model timing

behaviour of cyber-physical systems. Their aim is to prove safety, liveness and reliability properties. Toolkits that support capturing of timing parameters for e.g. schedulability analysis have been proposed by Stankovic (Stankovic 2001) and Singhoff et al. (Singhoff et al. 2005), among others. With these models it is possible to express bounds for occurrence of pseudoperiodic events. However, they are not able to model pseudoperiodic behaviour in more detail.

## 7.2. Cyber-Physical System Model

Computations in CPS can consist of multiple components that interact among each other to control different aspects of the physical system. These aspects may have different requirements for reactions from the cyber system, thus introducing different timing requirements for the respective computation components. In the following, the structural model that underlies GTM is described. Also, the timing parameters that can be annotated in the model are discussed.

### 7.2.1. System Model

Figure 7.1 gives an overview of the system model that is the basis for GTM. Basically, the CPS consists of a *Cyber System* that interacts with the *Physical System*. Concerning the cyber system, three types of components can be distinguished: *Sensor components (SCs)* and *actuator components (ACs)* are able to interact with the physical system; *Computation components (CCs)* perform computations and thus connect sensors and actuators, but cannot interact directly with the physical world.

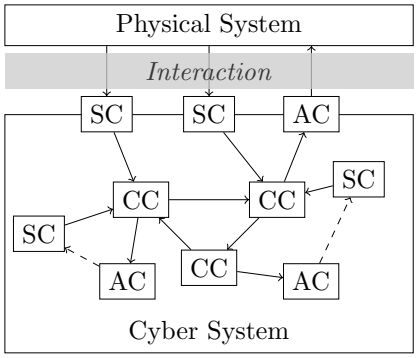



Figure 7.1.: Cyber-physical system model

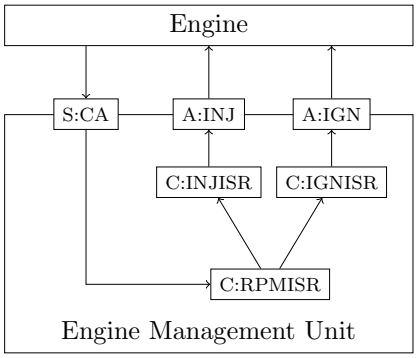



Figure 7.2.: Simplified model of FreeEMS example

Most *sensor components* translate from the physical world into the cyber world, but are not restricted to this. It is also possible to have SCs that monitor the behaviour of computations. SCs provide input data for computations. They play an important role in triggering computations. *Actuator components* form the counterpart to sensors. They translate computation results into physical states. Thus, they can directly impact on the environment or the CPS. They may also influence the behaviour of CPS-internal components. *Computation components* reside completely in the cyber world. They use SC and ACs to interact with the physical world. CCs perform computations on input data and generate output data. They

can exchange data among each other depending on the structure of the control algorithm they implement.

Components use *communication channels* to exchange data. These are indicated by arrows in figure 7.1. Communication usually takes place within the CPS, but might also be influenced by the environment (e.g. when using radio).

**Example 7.2.** *Figure 7.2 depicts a simplified view of the FreeEMS code (see ex. 7.1). A rotary sensor (S:CA) mounted to the crank shaft of the engine triggers the computation component* `PrimaryRPMISR` *(C:RPMISR in fig. 7.2). This component calculates the times and parameters for injection and ignition, and sets appropriate timers. These timers then trigger the actual injection (via C:INJISR and A:INJ) resp. ignition (via C:IGNISR and A:IGN) of fuel, where the parameters calculated by the* `PrimaryRPMISR` *are consumed. The actual implementation of FreeEMS has a much higher complexity. However for the aims of this chapter, the functionality described above suffices.*

Several types of timing parameters for a CPS can be distinguished: On the one hand, there are timing *requirements* that arise from the system's specification and propagate down to the single components. On the other hand, the physical and cyber parts have certain timing *properties* that restrict the CPS's implementation.

### 7.2.2. Timing Properties

Any component in the system model can and most certainly will have certain properties concerning its timing behaviour. Physical components, i.e. sensors and actuators, can have response times that establish natural bounds on how often their service can be requested. Communication channels can introduce transmission latencies or jitter that may lead to inconsistent data.

The timing properties of the cyber system are determined by the execution times of the computation components. Typical parameters are worst- and best-case execution times (WCET, BCET), or an execution time distribution function. These can be used to determine the jitter an output signal might suffer through computations. This information, in turn, can be used to adjust, e.g. the implementation of control algorithms to ensure a predefined quality of control.

A CPS interacts with its environment. This means that the behaviour of the environment influences the CPS and its timing behaviour. If aspects of the environment's behaviour are known, they should also be captured using the GTM formalism.

### 7.2.3. Timing Requirements

Timing requirements arise from the specification of a CPS. The design of the CPS must ensure that these timing requirements can be fulfilled through the different components and their timing properties.

The cyber and physical parts of a CPS usually form a control loop. Reactions in the cyber part are triggered through time or certain events. In the following, the term *trigger event* is used to embrace both possibilities. Trigger events can originate from the environment as well as from the physical or cyber parts of the system. Clearly, the central timing requirement is that a reaction to any trigger event happens within a certain time after the trigger event. The term *set event* is used to describe the point in time the computed reaction is set at the

actuator. Each set event relates to at least one earlier trigger event. Trigger events may relate to former trigger or set events (see sect. 7.3.1, def. 7.2).

## 7.3. The Generic Timing Model

The generic timing model introduces a formalism to capture parameters of CPSs that relate to the systems' timing behaviour. The goal of GTM is (1) to create a formal model of timing properties based on the structure and behaviour of a CPS, and (2) to provide generic means that enable developers to state requirements on the CPS's behaviour. Additionally, GTM provides helpers for the definition of utility functions that can be used, e.g. for scheduling. In this section, the basic notations of the GTM are defined.

### 7.3.1. Basics

The letter $c$ denotes time spans, and $t$ denotes concrete points in time. The time domain is represented by $\mathbb{T}$; it can be either $\mathbb{N}_0$ or $\mathbb{R}_0^+$. Undefined values are written as $\perp$. Based on these, the notion of events is defined:

**Definition 7.1.** *An event is a tuple* $e = (t_e, t_a)$, $t_e \in \mathbb{T} \cup \{\perp\}$, $t_a \in \mathbb{T}, t_e \leq t_a$ *of* expected *and* actual arrival times*. If an event of the same type recurs, it is written as a sequence* $E = (e_n)$ *with* $t_{x,i+1} > t_{x,i}\ \forall i \in \mathbb{N}_0, x \in \{e, a\}$*. The type of an event is determined through the event's source.*

For any event, its actual arrival time $t_a$ can be determined as soon as the event happens. For some types of events, it is possible to predetermine an expected arrival time $t_e$ from some knowledge about the system, which is e.g. the case for periodic timers. If this cannot be done, the expected arrival time is undefined, i.e. $t_e = \perp$.

**Example 7.3.** *The ISR* `RTIISR()` *in FreeEMS is triggered by the real-time interrupt (RTI) of the underlying hardware. The RTI controller generates an interrupt each* $p_{RTI} = 125\,\mu s$*. If the the first interrupt request (IRQ) is generated at time* $t_0$*, the expected arrival times of subsequent IRQs are given by* $t_{e,i} = t_0 + p_{RTI}$*. If an IRQ is asserted immediately, then* $t_{a,i} = t_{e,i}$*. However, if the acceptance of interrupts in the processor is deactivated at time* $t_{e,i}$ *(e.g. due to another IRQ being active), the IRQ is delayed and thus* $t_{a,i} > t_{e,i}$*.*

A reaction in the CPS can be characterised by two events, one that triggers the reaction, and another one that is raised when the result is set:

**Definition 7.2.** *A* reaction $r$ *is described by its trigger event* $e^t$ *and set event* $e^s$*,* $r = (e^t, e^s) = \big((t_e^t, t_a^t), (t_e^s, t_a^s)\big)$*. A function* $f_r$ *must exist that causally relates the set event to the trigger event, i.e.* $f_r : e^t \mapsto e^s$*. The definition of* $f_r$ *must preserve causality, i.e.* $t_e^t < t_e^s$*, and/or* $t_a^t < t_a^s$ *if* $t_e^t = \perp$*. Like events, reactions can recur sequentially as a reaction sequence* $R = (r_n)$*.*

Reactions in this sense can be applied to single components of the CPS and to chains of components as well. The reaction function $f_r$ that relates trigger and set event is constructed from the specification of the component that performs the reaction. Reactions can also be used to model the behaviour of (physical) communication channels.

**Example 7.4.** *One reaction in FreeEMS is the* `IgnitionFireISR()` *that triggers fuel ignition. When a preset timer expires, it yields a trigger event $e^t$ which leads to the execution of the ISR. In the end, the ISR sets the pin outputs and thus yields a set event $e^s$. The ISR itself represents the reaction function $f_r$ that relates trigger and set event.*

Some correctness metrics use a sliding window e.g. over the last $n$ events or reaction. Such behaviour can be found e.g. in scheduling of $(m, k)$-firm real-time tasks (Ramanathan 1999) or weakly-hard real-time system (Bernat et al. 2001). This can be formalised through connected subsets of sequences:

**Definition 7.3.** *Let $X = (x_n) = (x_0, x_1, \ldots)$ be a (possibly infinite) sequence. A* connected subset *of $X$ is denoted as*

$$X|_{p,q} = (x_p, x_{p+1}, \ldots, x_{q-1}, x_q)$$

*with $p, q \in \mathbb{N}_0 \cup \{\infty\}, p \leq q$. The whole sequence $X$ can be written as $X|_{0,\infty}$, a tail of the sequence as $X|_{p,\infty}$. Additionally, let*

$$\mathcal{S}(X) = \{X|_{p,q} | p, q \in \mathbb{N}_0, p \leq q\} \cup \{X, \emptyset\}$$

*be the set of all connected subsets of a sequence $X$.*

The utility metrics presented later in this chapter map events or reactions to a utility domain $\mathbb{U}$:

**Definition 7.4.** *$\mathbb{U} := [0, \mathcal{M}] \cup \{-\infty\}$ with $\mathcal{M} > 0$ denotes the utility domain. $\mathcal{M}$ means maximum benefit for the system, 0 means no benefit, and $-\infty$ stands for a failed computation with possibly catastrophic consequences. A function $u : \mathbb{D} \to \mathbb{U}$ is called* utility function for single reactions *with $\mathbb{D}$ being an event sequence or reaction sequence. A function $U : \mathcal{S}(\mathbb{D}) \to \mathbb{U}$ is called* utility function for a sequence of events resp. reactions *or* history-cognisant utility function (HCUF).

For simplicity, the following definitions assume $\mathcal{M} = 1$. Table 7.1 gives an overview of the parameters used throughout the following sections.

### 7.3.2. Reactions and their properties

Viewed from the outside, a CPS reacts to events occurring in the environment. This property of the whole CPS can be conveyed to any of its components (see sect. 7.2.1) that add to a CPS's reaction. Reactions in CPS usually do not happen on their own accord, but are triggered by events. Events may originate in the environment of the CPS, but can also be generated inside the CPS. Regardless of their origin, events are transmitted in only one way:

**System Property 7.1.** *Events are propagated over communication channels. The GTM allows multiple event types flowing through one channel as well as one event type flowing through multiple channels.*

Reactions in a CPS are described in the following way:

**System Property 7.2.** *A component $C$ in a CPS generates a reaction sequence $R_C$. Each reaction in $R_C$ is triggered by distinct events of the same type, and generates again distinct events of another type. Sensor components do not require CPS-internal trigger events. Instead, they are used to generate trigger events for any connected CPS components. Similarly, actuator components act as sinks for set events.*

Table 7.1.: Overview of notations used in GTM

| Symbol | Meaning |
|---|---|
| $c$ | execution time span |
| $c^B, c^W$ | best- and worst-case execution time spans |
| $t$ | a point in time |
| $t_e, t_a$ | expected resp. actual time of a certain event |
| $t^t, t^s$ | time of a trigger resp. set event |
| $e_i$ | event in a event sequence $E = (e_n)$ |
| $e^t, e^s$ | trigger/set event of a reaction |
| $r_i$ | reaction in a reaction sequence $R = (r_n)$ |
| $u$ | utility function for a single reactions |
| $U$ | history-cognisant utility function for a sequence of reactions |
| $z$ | counters |

Reaction sequences can be classified in two groups, containing *depending* or *isolated* reactions. The first group includes those reaction sequences, where each single reaction depends on the prior occurrence of another reaction. In GTM this is modeled with the following property:

**System Property 7.3.** *Let $R_D$ be a sequence of reactions $r_i$. $R_D$ is called* depending *on another sequence of reaction $R'_D$, if a* dependency function *$f_D$ can be derived from functional specification of the CPS such that $f_D$ relates the expected arrival time of the trigger event ($t^t_{e,i}$), or set event ($t^s_{e,i}$) of reactions $r_i \in R_D$ to the occurrence of reaction $r'_k \in R'_D$:*

$$\forall\, r_i \in R_D, i \geq 0\ \exists\, r'_k \in R'_D k \geq 0 : t^t_{e,i} = f_D(r'_k)\ \textit{or}\ t^s_{e,i} = f_D(r'_k) \tag{7.1}$$

A special case of this behaviour appears frequently for periodic reactions, when $R'_D = R_D$ and $r'_k = r_{i-1}$, i.e. a reaction depends on its predecessor.

**Example 7.5.** *The FreeEMS component* `PrimaryRPMISR()` *is triggered by certain angularities of the crank shaft (see ex. 7.2). One of its tasks is to set timers for fuel ignition, which is controlled by two ISRs* `IgnitionDwellISR()` *and* `IgnitionFireISR()`*. Each reaction of one of the ignition ISRs thus depends on a reaction performed by the* `PrimaryRPMISR()`*.*

The second group includes reaction sequences where the knowledge available within the CPS does not allow to relate a single reaction to any other reaction. Such *isolated reactions* are described by the following property. They can be used to model, e.g. a user pressing an input button.

**System Property 7.4.** *In an* isolated reaction *$r$, the trigger event has only an actual arrival time $t^t_a$, but the expected arrival time cannot be predetermined, i.e. $t^t_e = \perp$. The reaction function $f_r$ is defined such that $t^s_e = f_r(t^t_a)$.*

An isolated reaction $r = (e^t, e^s)$ starts with a trigger event $e^t = (\perp, t^t_a)$ and finishes with a set event $e^s = (t^s_e, t^s_a)$. It is important to note that although $t^t_a$ is arbitrary, once the event happened it is fixed and not subjected to any jitter. However, the value of $t^s_a$ and thus the

actual reaction time $c_r = t_a^s - t_a^t$ can be subject to jitter in computation and transmission of data, thus introducing *output jitter*.

Regardless to which group a reaction belongs, its proper execution will contribute to the overall benefit of the CPS. In real-time systems, the amount contributed to the system benefit significantly depends on the time when a reaction is completed. The GTM employs the well-known concept of time-utility functions (TUFs) (Jensen et al. 1985; Ravindran et al. 2005) to model this behaviour:

**System Property 7.5.** *Let $R = (r_n)$ be a sequence of reactions. A function*

$$\begin{array}{rccc} u_D: & R & \to & \mathbb{U} \\ & (e_i^t, e_i^s) & \mapsto & u_D(e_i^t, e_i^s) \end{array} \tag{7.2}$$

*is called the* delay utility *of a single reaction $r_i \in R$, if at least*

$$u_D\left((t_{e,i}^t, t_{a,i}^t), (t_{e,i}^s, t_{a,i}^s)\right) = 1 \quad \forall\, t_{a,i}^s = t_{e,i}^s \tag{7.3}$$

*This means that if the set events arrive at their expected arrival times the reaction has full benefit for the system. Reactions $r_i$ with $u_D(r_i) < 0$ or $0 \leq u_D(r_i) < 1$ are considered as lost resp. delayed. Depending on the actual system, also reactions that are finished too early may have negative impact on the systems which is also mapped through $u_D(r_i) < 1$. Any reaction $r_i$ with $u_D(r_i) \neq 1$ is considered to* deviate *from the desired behaviour.*

**Example 7.6.** *Real-time computing uses the notion of a deadline $d$ that specifies the time until which a computation has to be finished. Depending on the type of application, deadlines can be hard, firm or soft.*

*A hard deadline $d$ relative to the actual trigger time $t_a^t$ of a reaction specifies a reaction function $f_r$ (see def. 7.2) that derives the expected time $t_e^s$ of the set event in the following manner:*

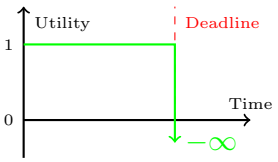
$$t_e^s = t_a^t + d \tag{7.4}$$

*If the set event actually arrives within the time span $d$, the reaction yield full utility to the system, else the consequences are catastrophic. This is represented by the following utility function (see also fig. 7.3(a)):*

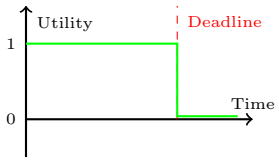
$$u_D(r) = \begin{cases} 1 & \text{if } t_a^t \leq t_e^t \\ -\infty & \text{if } t_a^t > t_e^t \end{cases} \tag{7.5}$$

*In a similar manner, TUFs for firm and soft deadlines can be specified (see fig. 7.3(b) and 7.3(c)).*

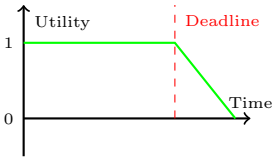
**Example 7.7.** *Concerning the actual ignition time, one can devise a delay utility function $u_D$ like the one depicted in figure 7.3(d). The EMS aims to trigger the ignition at $t_{\text{opt}}$ to achieve $u_D(r^{\text{ign}}[t_a^s = t_{\text{opt}}]) = 1$. Through interpolation from the ignition map, the EMS can only approximate $t_{\text{opt}}$. Untimely ignition can result in knocking of the engine, hence leading to higher abrasion. Late ignition can reduce engine efficiency. If ignition is delayed way too far, even catastrophic consequences in means of damages to the car could follow. All these circumstances influence $u_d$ and thus $t_{\text{opt}}$.*



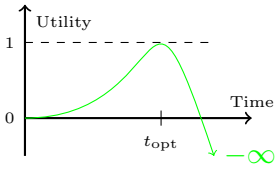

(a) Hard real-time job




(b) Firm real-time job




(c) Soft real-time job




(d) Fuel ignition

Figure 7.3.: Exemplary utility functions

### 7.3.3. Component Properties

Certain applications can tolerate a component missing some of its deadlines, i.e. some reactions being late. Nevertheless, they require a certain overall behaviour of the component over time. The delay utility defined above allows only to rate single reactions. To rate reaction sequences, the concept of a utility function can be extended to a component utility functions $U$:

**System Property 7.6.** *The overall contribution of a component to the system's benefit is measured by a history-cognisant utility function $U$ that works on reaction subsequences:*

$$\begin{array}{rccl} U: & \mathcal{S}(R) & \to & \mathbb{U} \\ & R|_{p,q} & \mapsto & U(u_D(r_p), \ldots, u_D(r_q)) \end{array} \tag{7.6}$$

A concrete definition of $U$ depends on the properties of the application. Simple history-cognisant utility functions can be defined in the following way:

**Example 7.8.** *One way to define the utility of a component is to use the minimum delay utility that occurred in a subsequence of reactions of the component:*

$$U_M(R|_{p,q}) = \min_{p \le i \le q} \{u_D(r_i)\} \tag{7.7}$$

*Another possibility is to calculate the average utility $U_A$ of a subsequence:*

$$U_A(R|_{p,q}) = \frac{\sum_{i=p}^{q} u_D(r_i)}{q - p + 1} \tag{7.8}$$

It is also possible, to define more complex history-cognisant utility functions that can describe the behaviour of a component in more detail. Therefore, counters are used:

**Definition 7.5.** *Let $R$ be a sequence of reactions. A counter $z$ for some formal property $P$ is a function*

$$\begin{array}{rccl} z: & \mathcal{S}(R) & \to & \mathbb{N}_0 \\ & R|_{p,q} & \mapsto & |\{r_i \in R|_{p,q}, r_i \textit{ fulfills } P\}| \,. \end{array} \tag{7.9}$$

Counters can be used to classify a component's behaviour under various facets. In certain use cases, it might be necessary to count reactions that (consecutively) do (not) underrun a lower utility bound $b_u$.

**Example 7.9.** *The number of reactions that underrun a lower utility bound $b_u$ within a subsequence of reactions can be determined as:*

$$z_D(R|_{p,q}) = |\{r_k | p \leq k \leq q \wedge 0 \leq u_D(r_k) < b_u\}| \tag{7.10}$$

The definition of history-cognisant utility functions based on such counters strongly depends on the system requirements. Therefore, examples for such functions are deferred to the following section.

### 7.3.4. System Requirements

Based on the properties introduced above, several templates are presented that can be used to model timing requirements on a system. The first requirement considers single reactions:

**System Requirement 7.1.** *If each reaction $r$ of a component must contribute to the system benefit, this is modeled in GTM by requiring that $u(r)$ never underruns a lower bound $b_u$:*

$$u(r) \geq b_u \quad \forall\, r \in R \tag{7.11}$$

**Example 7.10.** *Recall example 7.6, that presented a delay utility function for hard real-time reactions. The requirement that a deadline in a sequence $R_H$ of hard real-time reactions $r_h$ must never be missed can be expressed as*

$$u(r_h) \geq 1 \quad \forall\, r_h \in R_H \tag{7.12}$$

Certain systems have less strict requirements and can tolerate that single deadlines are missed (e.g. Hamdaoui and Ramanathan 1995; Bernat et al. 2001). This can be expressed by requiring that equation (7.11) holds only for *some* reactions $r \in R$. However, such system still require some guaranteed kind of behaviour. This can be captured by using the counters introduced in section 7.3.3 that count deviations from requirement 7.1:

**System Requirement 7.2.** *Let $z_d$ be a counter that counts the occurrence of* desired *timing behaviour in the reactions of a component. Further, let $z_u$ be a counter that counts occurrences* undesired *behaviour of a component. System requirements can state that these counters never under- resp. overrun certain bounds:*

$$z_d(R|_{p,q}) \geq z_d^{\min} \tag{7.13}$$
$$z_u(R|_{p,q}) \leq z_u^{\max} \tag{7.14}$$

**Example 7.11.** *Recall the counter $z_D$ defined in example 7.9. The system requirements can specify a maximum value $z_D^{\max}$ that shall not be exceeded within any time window of $h$ consecutive reactions:*

$$z_D(R|_{p,q}) \leq z_D^{\max}\ \forall p, q \in \mathbb{N}_0, q - p < h \tag{7.15}$$

Based on these requirements, one define an exemplary HCUF:

**Example 7.12.** *The following HCUF requires a positive time interval $h > 0$. It works on subsequences of reactions $R|_{p,q}$ with $q - p < h$. A HCUF for the delay counter requires that $z_D^{\max} > 0$, else this counter would not make much sense. Then it can be defined via the ratio of actual and permitted deviations:*

$$U_D(R|_{p,q}) = 1 - \frac{z_D(R|_{p,q})}{z_D^{\max}} \tag{7.16}$$

Such or similar HCUFs can be used to characterise the behaviour of a system component. Their use for scheduling is investigated in chapter 8.

## 7.4. Periodic and Pseudoperiodic Behaviour

In many CPSs, recurring dependent events, as defined in property 7.3, occur not randomly, but in a periodic manner. Formally, this means that $t_{e,i} = t_{e,i-1} + \pi_0$ with a constant $\pi_0$ for some event sequence $E = (e_n)$. For such strictly periodic behaviour, the formalism presented above can be applied directly. Jitter in such behaviour is captured by the formalism as well as through the delay utility function $u_D$ from property 7.5. However, it can also happen that the behaviour changes over time. In the short term, the system may expose a strictly periodic behaviour, but in the long term, the phase or frequency may change. A phase change manifests itself by one event $e_k$ arriving earlier or later than expected with an offset $\phi$ at time $t_{e,k} = t_{e,k-1} + \pi_0 + \phi$. The subsequent events then arrive again strictly periodically with respect to $e_k$ at times $t_{e,k+i} = t_{e,k} + i\pi_0$. Figure 7.4 illustrates this behaviour as a phase shift between the events at times $t_3$ and $t_4$.

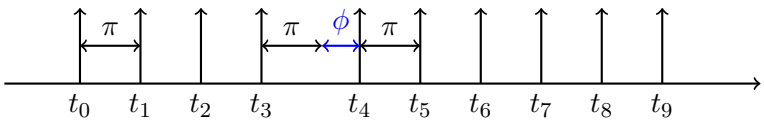



Figure 7.4.: Phase shift in periodic behaviour

**Definition 7.6.** *Let $E$ be an event sequence with initial period $\pi_0$ and sporadic phase changes, i.e. most events $e_i = (t_{e,i}, t_{a,i})$ adhere to the strict period with respect to their predecessor, $t_{e,i} = t_{e,i-1} + \pi_0$. Though, some events arrive deviating from this period and thus shift the arrival times of their successors, $t_{e,i} = t_{e,i-1} + \pi_0$. The sequence $\Phi = (\phi_n) \in \mathbb{T}$ with*

$$\begin{aligned} \phi_0 &= t_{e,0} \\ \phi_i &= t_{e,i} - t_{e,i-1} - \pi_0 \quad \forall i > 0. \end{aligned} \tag{7.17}$$

*denotes the* phase change sequence *for $E$.*

Please note that $\phi_i \neq 0$ only if a phase change occurs at event $e_i$. From this definition follows that for a known phase shift sequence $(\phi_n)$ the expected arrival times of events can be calculated as

$$t_{e,i} = i\pi_0 + \sum_{k=0}^{i} \phi_k \tag{7.18}$$

**Example 7.13.** *Consider a very simple hardware timer which can raise an interrupt if a counter expires and must be restarted by software, e.g. during the associated ISR. In this scenario, a new counting period starts after the ISR has reset the timer. If no variances in the ISR execution time are assumed, the timer would generate a strictly periodic event sequence with period $\pi$. However, the start of the ISR may be delayed due to another ISR being active. In this case, also the restart of the timer is delayed, introducing a positive offset $\phi_i$.*

Changes of the frequency, i.e. the period length, are slightly more complex to describe. A single frequency change in an event sequence $E$ appears, if two disjoint subsequences $E|_{0,q_0}, E|_{p_1,\infty}$ with the following properties can be found: (1) Events in $E|_{0,q_0}$ appear strictly periodic with distance $\delta_0$, i.e. $t_{a,i-1} - t_{a,i} = \delta_0 \; \forall t_{a,i-1}, t_{a,i} \in E|_{0,q_0}$; (2) Events in $E|_{p_1,\infty}$ appear strictly periodic with distance $\delta_1 \neq \delta_0$; either (3a) $q_0 = p_1$, i.e. $E|_{0,q_0}$ passes directly into $E|_{p_1,\infty}$, or (3b) $q_0 < p_1$ and the distance of events in $E|_{q_0,p_1}$ gradually changes from $\delta_0$ into $\delta_1$. This concept can easily be extended to multiple frequency changes. Figure 7.5

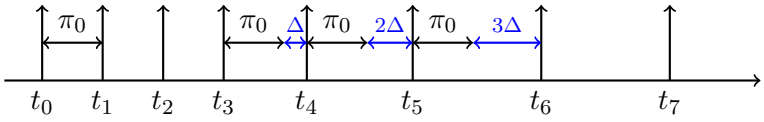



Figure 7.5.: Frequency change in periodic behaviour

illustrates such behaviour exemplarily. The frequency change from $\pi_0$ to $\pi_0 + 3\Delta$ gradually takes place in the intervals between $t_3$ and $t_6$. Formally, a *frequency change sequence* describes the process of a frequency change:

**Definition 7.7.** *Let $E$ be an event sequence with frequency changes as described above. Define a* frequency change sequence $\Pi(E) = (\pi_n)$ *where the $\pi_i$ describe the change in distance between event $e_{i+1}$ and $e_i$ with respect to the distance between $e_i$ and $e_{i-1}$:*

$$\begin{aligned} \pi_0 &= \textit{initial period} \\ \pi_i &= (t_{e,i+1} - t_{e,i}) - (t_{e,i} - t_{e,i-1}) \quad \forall\, i \geq 1 \end{aligned} \tag{7.19}$$

If the first event $e_0$ in a series $E$ is expected to occur at time $t_{e,0} = \phi_0$, and $\Pi(E)$ is known, the expected arrival times of subsequent events $e_{i+1}$ in $E$ can be calculated as:

$$t_{e,i+1} = t_{e,i} + \sum_{k=0}^{i} \pi_k = \phi_0 + \sum_{j=0}^{i} \sum_{k=0}^{j} \pi_k \tag{7.20}$$

Similar to phase changes, $\pi_i = 0$ for terms of constant frequency, and $\pi_i \neq 0$ only if a frequency change happens after an event $e_i$. Phase and frequency changes can also be combined:

$$t_{e,i+1} = \sum_{j=0}^{i} \phi_j + \sum_{j=0}^{i} \sum_{k=0}^{j} \pi_k \tag{7.21}$$

An event or reaction sequence is denoted as behaving *pseudoperiodic* if at least one of the above definitions 7.6 and 7.7 can be applied to its behaviour.

Although the actual $\phi_i$ and $\pi_i$ are usually unknown, they can be used to specify further properties or requirements of a CPS by bounding their values. The notion of phase shifts allows to formulate a timing behaviour requirement for a CPS by defining phase shift bounds $\phi_{\min}, \phi_{\max}$ for an event sequence $E$. The requirement then states that if events in $E$ occur with a phase shift within these bounds, i.e. $\phi_{\min} \leq \phi_i \leq \phi_{\max}\ \forall i \geq 0$, the CPS should gain full benefit. Only if event phase shifts occur that are outside these bounds, the system suffers a utility degradation.

Based on the frequency change sequence $\Pi$, properties of certain physical systems can be specified more precisely. One can define bounds $p_{\min}, p_{\max}$ for the frequency that shall or will not be exceeded, i.e. $p_{\min} \leq \sum_{j=1}^{i} \pi_j \leq p_{\max}\ \forall i \geq 1$. Furthermore, it is also possible to constrain how fast the frequency can change by constraining the single $\pi_i$ through $\pi_{\min} \leq \pi_i \leq \pi_{\max}\ \forall i \geq 1$.

**Example 7.14.** *Recall the* `PrimaryRPMISR()` *of FreeEMS that is triggered by the crank shaft sensor (ex. 7.2). The sensor generates an event sequence $E_{CA}$, which describes a pseudoperiodic process with frequency changes (described by a sequence $\Pi(E_{CA})$), as the rotation speed of the crank shaft can vary over time, but may also be constant for certain time intervals. However, several constraints on $E_{CA}$ can be derived: (1) There are lower and upper bounds for the engine speed and thus intervals between succeeding events in $E_{CA}$ are also upper and lower bounded. (2) Due to physical constraints, the engine speed cannot change abruptly, but only gradually. Thus, $\pi_{\min}, \pi_{\max}$ can be found that bound all $\pi_i \in \Pi(E_{CA})$. These bounds could be used to predict how much guaranteed processing time will be available in the short-term future for computations other than EMS interrupt handling.*

What these parameter definitions do not allow yet is to differentiate between jitter and changes of phase or period. This could affect the delay utility $u_D$ (see property 7.5) if it is calculated at runtime. In this case, the CPS must provide additional information e.g. through sensors that allow such a differentiation.

## 7.5. Summary

GTM provides a formalism to model the timing behaviour of CPSs and their interaction with physical processes. It can map physical (timing) properties and system requirements into the development process. The GTM formalism can be employed in a hierarchical system description, ranging from high-level specifications down to the actual component implementations. Additionally, it comprises a method to capture pseudoperiodic behaviour in a formal manner. Time utility functions and history-cognisant utility function based on the timing parameters enable a flexible description of the required and actual timing behaviour of a CPS. GTM utility functions are solely based on timing requirements. However, it is possible to combine these functions with other approaches that measure how much value a certain component adds to the system's benefit (cf. Burns et al. 2000).

TUFs and HCUFs provide a means for flexible scheduling of real-time tasks, especially in systems that tolerate sporadic deadline misses, e.g. due to temporary overloads. This topic is discussed further in chapter 8. The GTM formalisms can be used to derive timing constraints for the execution behaviour. These constraints can be used to strengthen the runtime coordination between real-time tasks. An outlook on this topic can be found in chapter 9, where also the exploitation of pseudoperiodic behaviour is discussed.

# 8

# Task Sets with Relaxed Real-Time Constraints

*Parts of this chapter are based on Kluge et al. 2015; an abridged version of this chapter has been submitted to* LITES: Leibniz Transactions on Embedded Systems

Applying hard real-time schedulability analysis on a task set can often lead to a pessimistic dimensioning of the underlying system. To ensure that task deadlines are kept in any case, schedulability analysis must take each task's WCET bound as base. Safe WCET bounds are derived through a static WCET analysis. Due to overestimations and pessimistic assumptions in the analysis, the WCET bound for a task may exceed measured and actual execution times by far. Nevertheless, the execution platform must be dimensioned for this WCET bound, as only in this case hard guarantees can be given. Thus, during regular operation within the observed execution time bounds, resources in terms of processing time and energy are wasted. One way to tackle this problems builds on special applications: Certain applications can be designed such that they can tolerate sporadic deadline misses. This allows to design a system based on less pessimistic execution time estimates that are, e.g. derived through measurements. Cases where these parameters are too optimistic, i.e. jobs execute longer, manifest themselves as an overload condition. These can be resolved either by deferring or cancelling some jobs.

Consider, for example, the decoding of a video stream. If single frames are displayed too late, the quality a viewer experiences degrades, but he stills can draw some benefit. Similarly, control systems can also tolerate some job losses due to their robustness (Bhattacharyya et al. 1995). Obviously, applying a simple quality of service (QoS) metric on the ratio of successful task executions is not sufficient, as it cannot prohibit that deadline misses happen too often consecutively. Specialised concepts have been developed in scheduling theory that allow to restrict distribution of deadline misses, for example the skip-over model (Koren and Shasha 1995), $(m, k)$-firm real-time tasks (Hamdaoui and Ramanathan 1995), the dynamic window-constrained scheduler (West and Schwan 1999), or weakly-hard real-time tasks (Bernat et al. 2001).

To convey the notion of systems with relaxed real-time constraints into real-time scheduling, Jensen et al. (Jensen et al. 1985) and Locke (Locke 1986) replaced the binary notion of a deadlines with more expressive TUFs and proposed an EDF-based scheduler. A TUF describes the value or utility a system can draw from a job execution if it is finished until a certain time, thus increasing the flexibility of real-time systems. A problem in TUF-based real-time scheduling is that each job is viewed independently. Therefore, no guarantees can be given about the distribution of deadline misses or job cancellations (both termed *losses*

in the following) for single tasks. It may even happen that jobs of a specific tasks are never executed (Kluge et al. 2013).

The aim of this chapter is to combine the concepts of $(m, k)$-firm real-time tasks and TUF-based scheduling by employing history-cognisant utility functions (HCUFs, see def. 7.4 in sect. 7.3.1). In the following section, related work on $(m, k)$-firm real-time tasks, related task models, and TUF-based scheduling is reviewed. Approaches for scheduling of $(m, k)$-firm real-time tasks that are important for this chapter are detailed in section 8.2, along with some new properties so far not reported in literature. In section 8.3, the MKU scheduling heuristic and some results on schedulability are presented. The methodology used in experimental evaluations is introduced in section 8.4. Evaluation results are shown and discussed in section 8.5. A summary in section 8.6 concludes this chapter.

## 8.1. Related Work

### 8.1.1. $(m, k)$-Firm Real-Time Tasks

The concept of real-time tasks with $(m, k)$-firm deadlines was proposed by Hamdaoui and Ramanathan (Hamdaoui and Ramanathan 1995). The idea behind this model is that a task needs to meet at least $m$ deadlines in $k$ consecutive jobs. An $(m, k)$-firm real-time task is a tuple $\tau_i = (C_i, T_i, m_i, k_i)$ with execution time $C_i$, period $T_i$ and $(m, k)$-constraint $(m_i, k_i)$. A task's deadline is assumed to be equal to its period. An $(m, k)$-firm real-time task incurs a dynamic failure, if less than $m$ out of $k$ consecutive jobs keep their deadline. Thus, the model goes beyond a simple QoS metric in terms of the ratio of missed deadlines by additionally restricting the distribution of deadline misses over the life cycle of a task. In the original work, a scheme for distance-based priority assignment of newly generated jobs depending of a task's distance from dynamic failure was proposed. Goossens (J. Goossens 2008) points out several problems of the DBP approach and devises an exact schedulability test. The seminal work of Hamdaoui and Ramanathan (Hamdaoui and Ramanathan 1995) has sparked a number of further works on the scheduling of $(m, k)$-firm real-time tasks. Ramanathan uses the concept of $(m, k)$-firm real-time tasks for the specific use case of control systems (Ramanathan 1999). A deterministic classification into mandatory and optional jobs is proposed based on static $(m, k)$-patterns. He also presents an appropriate controller design. Quan and Hu (Quan and Hu 2000) improve schedulability by the introduction of rotation values for $(m, k)$-patterns to relieve critical instants.

Jia et al. (Jia et al. 2005) examine the generation of $(m, k)$-pattern in more detail. Based upon this work, they derive a sufficient schedulability condition for $(m, k)$-firm real-time task sets in control systems and present an algorithm that aims to find at least sub-optimal values for $m$ (Jia et al. 2007). Flavia et al. (Flavia et al. 2008) present an algorithm that dynamically assigns $(m, k)$-parameters for plant control. Cho et al. devise two schemes for *Guaranteed Dynamic Priority Assignment (GDPA)* (Cho, Chung, et al. 2010). Both schemes are based on EDF scheduling, but additionally take the tasks' distance from a failing state into account. They are aimed at (1) providing a bounded probability of violations of the $(m, k)$-firm constraints, and (2) maximising the probability of kept deadlines. In a later work, by Rhu et al. (Rhu et al. 2011) propose a similar scheme, called gMUA-MK, for scheduling on multiprocessor systems. Concerning non-preemptive scheduling of $(m, k)$-firm real-time tasks, the work on Matrix-DBP (Poggi et al. 2003) provides necessary schedulability conditions. Li et al. (J. Li et al. 2004) derive a sufficient schedulability condition for NP-DBP-EDF. Kong

and Cho (Kong and Cho 2011) present an algorithm for scheduling of $(m, k)$-firm real-time tasks on multiprocessors.

### 8.1.2. TUF-Based Real-Time Scheduling

The concept of time-utility functions was originally introduced by Jensen et al. (Jensen et al. 1985) and Locke (Locke 1986). Instead of basing task scheduling solely on the binary notion of a deadline, the use of TUFs allows for a greater flexibility. The benefit of TUFs is demonstrated on EDF scheduling of overloaded task sets. If a high probability for a deadline miss is detected that would render the EDF schedule infeasible, jobs that only contribute with a low value-density (ratio of utility/value to execution time) to the system are selectively cancelled. Thus, schedulability of the system is ensured and accumulated utility is maximised. In literature, this approach is often referred to as *Locke's Best-Effort Scheduling Algorithm (LBESA)*. Based on LBESA, Clark (Clark 1990) has developed the dependent activities scheduling algorithm (DASA) for tasks with dependencies. Aldarmi and Burns (Aldarmi and Burns 1999) proposed the notion of *dynamic value density (DVD)* to reduce cancellations of jobs that have already started execution. Li and Ravindran (P. Li and Ravindran 2004) presented the MLBESA and MDASA algorithms that mimic the behaviour of LBESA resp. DASA, but come with lower complexities. TUF-based approaches have often been proposed to handle transient overloads in real-time systems (Baruah et al. 1991; Koren and Shasha 1992; G. Buttazzo et al. 1995; Mosse et al. 1999; Mejía-Alvarez et al. 2000; Ding and Guo 2008).

Many works on scheduling based on TUFs can also be found under the term *utility-accrual (UA) scheduling*. The aim in UA scheduling is to maximise the utility that is accrued through the execution of tasks. Insofar, the values and shapes of TUFs are a central criterion for scheduling. Chen and Muhlethaler (K. Chen and Muhlethaler 1996) have shown that the problem of maximising value through arrangement of jobs/tasks is NP-hard. They also proposed an heuristic scheduling algorithm with a complexity of $O(n^3)$ that is often referred to as *Chen and Muhlethaler's algorithm (CMA)*. The UPA algorithm by Wang and Ravindran (J. Wang and Ravindran 2004) for packed scheduling in switched ethernet comes with a lower complexity of $O(n^2)$, but is restricted to unimodal nonincreasing TUFs. In contrast, the *resource-constrained utility accrual (RUA)* by Wu et al. (Wu et al. 2004) can handle arbitrary TUFs and resource constraints at a complexity of $O(n^2 \log n)$. The *generic utility scheduling (GUS)* algorithm by Li et al. (P. Li et al. 2006) can also deal with mutual exclusion constraints, although with higher complexities of $O(n^3)$ for dispatching and $O(n^4 r)$ for scheduling. Tidwell et al. (Tidwell et al. 2010) model the scheduling problem as a Markov decision process that is solved offline and yields an optimal solution. The solution is used to generate a lookup table that is evaluated by an online scheduler in linear complexity. Also, works exist that investigate the use of TUF-based scheduling on multiprocessor systems (Swaminathan and Manimaran 2002; Cho et al. 2006; Cho, Ravindran, et al. 2010; Rhu et al. 2011). Here, especially the work of Rhu et al. (Rhu et al. 2011) on the gMUA-MK algorithm is interesting, as they aim to schedule tasks with $(m, k)$-firm deadlines and TUFs on multiprocessors.

## 8.2. Scheduling of $(m, k)$-firm Real-Time Tasks

In this chapter, DBP, fixed-pattern based and preemptive EDF-based schedulers are used for comparison with MKU. DBP (Hamdaoui and Ramanathan 1995) is chosen because it yields a very good performance, as the results in section 8.5 will show, and because an exact schedulability test exists (J. Goossens 2008). For the pattern-based approaches (Ramanathan 1999; Quan and Hu 2000), a direct application to control systems exists, and a sufficient schedulability test is available (Jia et al. 2007). Both DBP and pattern-based approaches use a fixed-priority scheduler for job execution. As a representative for scheduling with dynamic priorities, the EDF-based approaches GDPA (Cho, Chung, et al. 2010) and gMUA-MK (Rhu et al. 2011) are used. In the following, the functionality of these approaches is described. The last part of this section points out some properties that apply to several of the approaches.

### 8.2.1. DBP Scheduling

For fixed-priority scheduling of $(m, k)$-firm real-time tasks, Hamdaoui and Ramanathan (Hamdaoui and Ramanathan 1995) propose a distance-based priority assignment scheme where the priority of a job depends on its task's distance from a dynamic failure, i.e. how many subsequent deadline misses can be tolerated until the task's $(m, k)$-constraint is violated.

Formally, DBP scheduling works as follows: An $(m, k)$-firm real-time task is a tuple $\tau_i = (C_i, T_i, m_i, k_i)$ with execution time $C_i$, period $T_i$, and $(m, k)$-constraint $(m_i, k_i)$. A task's deadline is equal to its period. The aim is to schedule a set of tasks $\tau = \{\tau_1, \ldots, \tau_n\}$ on a single processor such that for each task at least $m_i$ out of $k_i$ consecutive jobs keep their deadlines. Let $\sigma_i^j \in \{0, 1\}$ denote the status of the $j$-th job of $\tau_i$ with 0 representing a deadline miss or job cancellation, and 1 standing for successful execution. Then, $\tau_i$'s state or *$k$-sequence* after execution of the $j$-th job is a string $\sigma_i = (\sigma_i^{j-k+1}, \ldots, \sigma_i^{j-1}, \sigma_i^j)$ with $\sigma_i^l \in \{0, 1\}^{k_i}$. New job states $\sigma_i^j$ are shifted into $\sigma_i$ from the right. Obviously, the $(m, k)$-constraint requires that a task $\tau_i$'s $k$-sequence always contains at least $m_i$ 1s. A task's *distance* from dynamic failure is the number of jobs that consecutively would have to miss their deadlines such that the task's $(m, k)$-constraint is no longer kept. If a new job $\tau_{i,j}$ is released, it is assigned a priority that depends on $\tau_i$'s distance from dynamic failure. Therefor, Hamdaoui and Ramanathan first define a function:

$$l_i(n, \sigma_i) = \begin{cases} k_i + 1, \text{if } \sigma_i \text{ contains less than } n \text{ 1s} \\ \text{position of } n\text{th 1 from the right, else} \end{cases} \tag{8.1}$$

Then, a new job's priority $p_{i,j+1}$ is calculated as:

$$p_{i,j+1} = k_i - l_i(m_i, \sigma_i) + 1 \tag{8.2}$$

with the task's current state $\sigma_i = (\sigma_i^{j-k+1}, \ldots, \sigma_i^{j-1}, \sigma_i^j)$. Lower values for $p_{i,j}$ represent a higher priority for scheduling. Hamdaoui's and Ramanathan's evaluations show that this priority assignment yields a better performance than assigning each task a fixed priority.

In a publication from 2008 (J. Goossens 2008), Goossens points out two important properties of the DBP scheme. First, the initialisation of a task's $k$-sequence $\sigma_i$ is a non-trivial problem. Based on an example, Goossens shows that starting with $\sigma_i = 1^k$ may not be optimal and lead to an infeasible schedule. The same example shows furthermore that initialising $\sigma_i$ with an error state can yield a feasible schedule.

Second, Goossens devises an exact schedulability test for $(m, k)$-firm real-time tasks under DBP scheduling. The test is based on the property that any feasible DBP schedule is periodic. This follows from the fact that scheduling decisions only depend on the $(m, k)$-constraints and $k$-sequences $\sigma_i$ of the tasks, whose space is bounded. Let $P = \text{lcm}\{T_i \mid i = 1, \ldots, n\}$ be the hyperperiod of the task set $\tau = \{\tau_i = (C_i, T_i, m_i, k_i) \mid i = 1, \ldots, n\}$. Then, the period of a feasible DBP schedule is bounded by

$$F = \prod_{i=1}^{n} \sum_{j=m_i}^{k_i} \binom{k_i}{j} \times P \tag{8.3}$$

Thus, if a task set $\tau$ is feasible in the interval $[0, F)$, i.e. no $(m, k)$-constraint is violated, then $\tau$ is always feasible. The exact schedulability test for $\tau$ consist of executing or simulating $\tau$ for this time interval and checking whether the $(m, k)$-constraints of all tasks are always kept. Once an $(m, k)$-constraint is violated, the test stops and returns that $\tau$ is not feasible. The test can be sped up by evaluating the *system state* $\sigma = (\sigma_1, \ldots, \sigma_n)$ consisting of all tasks' $k$-sequences at each hyperperiod boundary $B = nP, n \in \mathbb{N}$. Jobs that were activated before this boundary have either finished execution, or were cancelled. Jobs activated at $B$ will be executed just after $B$. Concerning job execution, the system is in the same state at each $B$. Only the $(m, k)$-states $\sigma_i$ may differ. If a certain system state $\sigma$ recurs, simulation can immediately be stopped, as the schedule will repeat itself and thus is feasible. In the following, this optimised test is termed *Goossens' schedulability test (GST)*.

These properties open two problems. First, how should the initial $k$-sequence be chosen? Principally, $2^{\sum_{i=1}^{n} k_i}$ sequences would have to be checked, which is rather infeasible. Second, the actual complexity of the schedulability test is very high, as no guarantees can be given concerning the periodicity of a schedule. Goossens presumes at the end of his paper that the actual complexity of the off-line test will be reasonable if realistic task sets with restricted $(m, k)$-constraints are considered. Both problems will be tackled later in this chapter.

### 8.2.2. Fixed $(m, k)$-Patterns

For tasks scheduled with DBP, it is hard to predict which jobs are executed successfully, and which are cancelled due to deadline misses. This can have negative impact on a control law that is implemented in such tasks. Ramanathan (Ramanathan 1999) presents a deterministic policy for classifying jobs of $(m, k)$-firm real-time tasks into mandatory and optional instances. Mandatory jobs are scheduled with their original, e.g. rate-monotonic (C. L. Liu and Layland 1973) priority while optional jobs get the lowest possible priority. Assume that jobs $\tau_{i,j}$ of a tasks $\tau_i$ are activated at times $jT_i, j = 0, 1, \ldots$. $\tau_{i,j}$ is mandatory, if

$$j = \left\lfloor \left\lceil \frac{jm_i}{k_i} \right\rceil \cdot \frac{k_i}{m_i} \right\rfloor, \tag{8.4}$$

else, it is classified as optional. Equation (8.4) generates a fixed pattern of mandatory resp. optional instances, which repeats after $k_i$ jobs. This pattern is incorporated into the design of the control law that a task implements. A set of $(m, k)$-firm real-time tasks $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$ is considered feasible, if at least all mandatory jobs can be executed successfully. A schedulability test is based on the fact that equation (8.4) classifies the first instance $\tau_{i,0}$ of any task $\tau_i$ as mandatory. Ramanathan (Ramanathan 1999) provides a sufficient schedulability condition. However, it contains a timing non-deterministic term which makes it hard to evaluate

(Jia et al. 2007). Jia et al. (Jia et al. 2007) propose a schedulability test, which basically implements the response time analysis (Audsley et al. 1993) for the first job of any task heeding the $(m, k)$-patterns. For task sets with harmonic periods, the test provides exact results, for all other task sets it is only sufficient.

Quan and Hu (Quan and Hu 2000) note that the classification according to equation (8.4) introduces a high pessimism into the schedulability analysis, as at time $t = 0$ a mandatory job from any task in a task set gets ready. They relieve this critical instant by introducing spin or rotation values for the $(m, k)$-patterns. Thus, for each task $\tau_i$ in a set of $(m, k)$-firm real-time tasks, an additional *spin parameter* $s_i$ is introduced. $s_i$ is used in the classification of mandatory resp. optional jobs by adjusting equation 8.4. A job $\tau_{i,j}$ then is classified as mandatory, if

$$j + s_i = \left\lfloor \left\lceil \frac{(j + s_i)m_i}{k_i} \right\rceil \cdot \frac{k_i}{m_i} \right\rfloor \tag{8.5}$$

If the $(m, k)$-pattern for a task $\tau_i$ is represented by a string $\pi_i = (\pi_{i,0}, \pi_{i,1}, \ldots,$ $\pi_{i,k_i-1})$, $\pi_{i,j} \in \{0, 1\}$ where 0 stands for an optional and 1 for a mandatory job, then equation (8.5) effectively rotates $\pi_i$ by $s_i$ places to the left. In the same paper, Quan and Hu also show that the problem of finding optimal $(m, k)$-patterns such that a specific task set is feasible, is NP-hard. They propose a heuristic algorithm for finding good spin parameters, and also examine the use of a genetic algorithm for the determination of spin parameters. For the work presented in this chapter, the heuristic algorithm is used. As the original persentation of the algorithm in Quan and Hu 2000 is missing important information, the corrections described in (Kluge 2016) are applied. In the following, the terms MKP resp. MKP-S shall refer to scheduling based on the original $(m, k)$-patterns by Ramanathan (Ramanathan 1999) resp. the patterns with spin parameter devised by Quan and Hu (Quan and Hu 2000).

### 8.2.3. EDF-based Scheduling

The GDPA scheme by Cho et al. (Cho, Chung, et al. 2010) uses EDF (C. L. Liu and Layland 1973) for the calculation of a schedule. However, ready jobs are not directly inserted into EDF schedule, instead they are kept in ready list. Any time the schedule needs to be adjusted, each job's distance from dynamic failure is calculated (see eq. (8.1)). The EDF schedule is created by considering the jobs in increasing order of their distance from dynamic failure, i.e. critical jobs are preferred, and inserting them into the EDF schedule. If an insertion makes the schedule infeasible, the job is removed again from the schedule. Jobs in the ready list that are infeasible are cancelled.

In the same paper, the Simplified Guaranteed Dynamic Priority Assignment (GDPA-S) is proposed. It works similar to GDPA, but has a lower runtime complexity. GDPA-S keeps ready jobs in two lists, one in EDF order and another ordered in increasing distance from dynamic failure. Dispatching is performed either from the head of the EDF list, if the EDF schedule is feasible. Else, the most critical job (at the head of the second list) is dispatched. Again, infeasible jobs are cancelled immediately.

The gMUA-MK approach by Rhu et al. (Rhu et al. 2011) is also used in the evaluations presented in this chapter. If reduced to a single processor, gMUA-MK works similar to LBESA. It differs in only two aspects: (1) The calculation of the value-density additionally heeds a task's distance to dynamic failure (see eq. (8.1)), and (2) gMUA-MK does not cancel jobs, but only postpones them.

In all three of the above schemes, scheduling decisions are solely based on the tasks' static properties and their $k$-sequences. Thus, GST can be applied for these schemes.

### 8.2.4. Properties of $(m, k)$-Schedules

In the following, some properties of $(m, k)$-firm task sets are described. First, a necessary schedulability condition is recapitulated, and then two results are presented that have not yet been reported in literature.

#### Necessary Schedulability Condition

In the evaluations, one of the necessary schedulability conditions that was described by Poggi et al. (Poggi et al. 2003) is used. The minimum load that is generated by a set of $(m, k)$-firm tasks $\tau = \{\tau_1, \ldots \tau_n\}$ is calculated as:

$$U_{\text{mk}} = \sum_{i=1}^{n} \frac{m_i}{k_i} \frac{C_i}{T_i} \tag{8.6}$$

In the following, the term $(m, k)$-*utilisation* shall refer to $U_{\text{mk}}$. The calculation of $U_{\text{mk}}$ assumes that only $m_i$ out of $k_i$ jobs of any task $\tau_i$ are executed. Now, if

$$U_{\text{mk}} > 1 \tag{8.7}$$

the task set is not feasible. A proof for this condition can be directly obtained from the proof that $U = \sum_{i=1}^{n} \frac{C_i}{T_i}$ is a necessary condition for feasibility under earliest deadline first (EDF).

#### Feasibility of Approaches Based on Fixed $(m, k)$-Pattern

For $(m, k)$-firm real-time task sets that use fixed $(m, k)$-patterns defined in (Ramanathan 1999) (MKP), Jia et al. give a sufficient schedulability test (Jia et al. 2007). As the test is only sufficient, it may reject some task sets that are actually feasible. Also, it cannot be applied to the MKP-S (Quan and Hu 2000) approach, as the rotation of the patterns might move the critical instant of the task set. Nevertheless, an exact schedulability test can be derived for task sets that used fixed $(m, k)$-patterns. The derivation of this test is similar to the one for DBP scheduling (J. Goossens 2008) and uses the same preconditions: (1) The scheduling algorithm must be *deterministic*, and (2) it must be *memory-less*. In the context of fixed $(m, k)$-patterns, the second precondition means that the algorithm's decisions at any time depend only on static properties of the active tasks. In contrast to (J. Goossens 2008), the current $k$-sequence of a tasks has no influence on the schedule.

The exact schedulability test follows from the periodicity of schedules when using fixed $(m, k)$-patterns

**Theorem 8.1.** *Let a set of synchronous $(m, k)$-firm real-time tasks be scheduled by a fixed-priority scheduler. Priorities are derived from fixed $(m, k)$-patterns that classify jobs into mandatory and optional. Then the schedule is periodic with period*

$$P = \text{lcm}\{k_i T_i \mid i = 1 \ldots n\}. \tag{8.8}$$

*Proof.* The priorities of jobs of a single task $\tau_i$ are derived from a fixed $(m,k)$-pattern of length $k_i$. Thus, each $k_i$ jobs, i.e. after $k_iT_i$ cycles, the priority pattern recurs. For any two tasks $\tau_i, \tau_j$, the job and priority pattern generated by both recurs after $\text{lcm}\{k_iT_i, k_jT_j\}$ cycles, as after this times both tasks' are in the same state as at the beginning (concerning their patterns). Via induction, this argument can be extended to $n$ tasks $\tau_1, \ldots, \tau_n$. □

For a task set where the sufficient test by Jia et al. (Jia et al. 2007) does not indicate feasibility or where the test is not applicable (MKP-S), it suffices the simulate the schedule for at most $P$ cycles (eq. (8.8)).

### Breakdown Anomalies

Consider an abstract task set (ATS) where tasks have no execution time parameters $C_i$, but instead execution time weights $e_i$. From this ATS, concrete task sets (CTSs) can be derived if a utilisation $U$ is given. Then, the $C_i$ are calculated from the $e_i$ such that the resulting CTS has utilisation $U$. In *hard* real-time scheduling, it is possible to identify a *breakdown utilisation* (Lehoczky et al. 1989) for an ATS, beyond which the derived CTSs are no longer feasible. A formal description of this approach follows later in section 8.4.1.

This method does not yield exact results for $(m,k)$-firm real-time task sets. Increasing the utilisation of a $(m,k)$-firm task set farther beyond the breakdown utilisation can actually lead to the task set being feasible again. To make the point more clearly, consider an ATS $\alpha$ with a breakdown target utilisation $U_\text{B}$. This means that the derived CTS with target utilisation $U_\text{B}$ is feasible, but the CTS with target utilisation $U_\text{B} + s_\text{U}$ ($s_\text{U} > 0$) is not. However, it may happen that the CTS with target utilisation $U_\text{B} + 2s_\text{U}$ is feasible again.

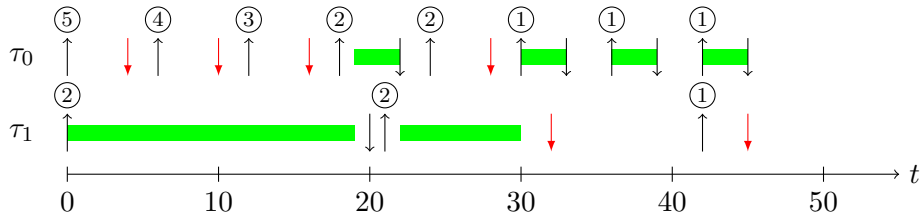
Table 8.1.: This ATS exhibits a breakdown anomaly at the target utilisations $U_\text{T} = 1.45$ and $U_\text{T} = 1.55$ when scheduled with DBP.

| Task | $T_i$ | $e_i$ | $(m,k)$ | $C_i^{U_\text{T}=1.45}$ | $C_i^{U_\text{T}=1.55}$ |
|---|---|---|---|---|---|
| $\tau_0$ | 6 | 55 | $(4,8)$ | 3 | 3 |
| $\tau_1$ | 21 | 95 | $(1,2)$ | 19 | 21 |

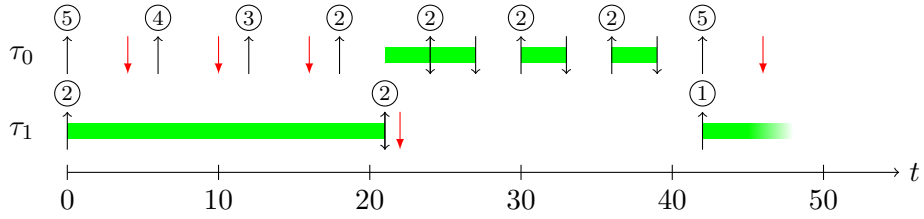
An exemplary ATS $\alpha$ that exhibits such a behaviour is shown in table 8.1, the corresponding schedules can be found in figure 8.1. The anomaly arises for target utilisations $U_T = 1.45$ and $U_T = 1.55$ (actual execution times are rounded to integer values). Figure 8.1(a) shows the schedule for $\tau(\alpha, 1.45)$. Numbers in circles indicate the distance-based priority of each released job. In time step 45, $\tau_{1,2}$ is cancelled, thus violating $\tau_1$'s $(1,2)$-constraint. Even if the scheduler cancelled the current instance of $\tau_0$ instead, the schedule would still be infeasible, as then $\tau_0$'s $(4,8)$-constraint would be violated later. Increasing $U_T$ for $\alpha$ leads to a feasible schedule, as is shown in figure 8.1(b). In this case, the segment between $t = 0$ and $t = 42$ is repeated periodically.

Such breakdown anomalies can occur when the scheduler has full flexibility in its decision on which jobs to cancel. They cannot happen in tasks that are scheduled using fixed $(m,k)$-patterns, which is expressed by the following theorem:

**Theorem 8.2.** *Let a set of $(m,k)$-firm real-time tasks $\tau^1$ be derived from an ATS $\alpha$ for a given utilisation $U_1$. Further, assume that $\tau^1$ is not feasible using fixed $(m,k)$-patterns. Then, any task set $\tau^2$ derived from $\alpha$ for a utilisation $U_2 > U_1$ is also not feasible.*




(a) $U_{\mathrm{T}} = 1.45$, Task 1 violates its (1,2)-constraint at time 45




(b) $U_{\mathrm{T}} = 1.55$, task set is feasible

Figure 8.1.: Example schedules with breakdown anomaly; for task parameters refer to table 8.1; circled numbers (ⓧ) denote DBP of job; red arrows (↓) indicate job cancellations.

*Proof.* Fixed $(m, k)$-patterns classify jobs into mandatory and optional jobs. A schedule $S$ is considered feasible, if all mandatory jobs are executed successfully. Infeasibility of $S$ means that at least one mandatory job $\tau_{i,j}$ misses its deadline at time $d_{i,j}$. Assume that $\tau_{i,j}$ is released at time $a_{i,j}$. Then, only mandatory jobs with priority higher than or equal to $\tau_{i,j}$'s priority are executed in the interval $[a_{i,j}, d_{i,j}]$. Increasing the utilisation of the task set means that the tasks' execution times are increased, but periods and thus activation times and deadlines remain unchanged. Thus, the processing time demanded by higher-priority jobs in $[a_{i,j}, d_{i,j}]$ can only further increase, and thus the deadline miss would occur also in the new task set. □

## 8.3. HCUF-Based Scheduling of $(m, k)$-firm Real-Time Tasks

The aim of this chapter is to study the feasibility of HCUF-based scheduling of $(m, k)$-firm real-time tasks. It is based on the task model already introduced in section 8.2.1, i.e. an $(m, k)$-firm real-time task is a tuple $\tau_i = (C_i, T_i, m_i, k_i)$. It is assumed that a task's actual execution time does not exhibit a great variability and approximates $C_i$, as e.g. is the case for control applications. Any task is released initially at time $t = 0$ and has an implicit deadline $D_i = T_i$. Thus, jobs $\tau_{i,j}$ are generated at times $r_{i,j} = jT_i, j = 0, 1, \ldots$ and must be finished until $d_{i,j} = (j + 1)T_i$, i.e. the task's relative deadline equals its period. The completion time of job $\tau_{i,j}$ is denoted as $f_{i,j}$. A task's $(m, k)$-constraint is defined by $(m_i, k_i)$, meaning that in any $k_i$ consecutively released jobs at least $m_i$ must be finished before their deadline. Furthermore, each job is subject to a firm real-time requirement: If the job is not finished before its deadline, its result is useless and the job is cancelled. The $(m, k)$-constraint given by $(m_i, k_i)$ states that from any $k_i$ consecutive jobs, at least $m_i$ must not miss their deadlines.

The aim of a scheduler is to execute a set $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$ of $n$ independent $(m, k)$-firm real-time tasks preemptively on one processor without violating any task's $(m, k)$-constraint.

In this section, first a history cognisant utility function is defined to map a task's current state in terms of its $(m,k)$-constraint. Then, the heuristic MKU algorithm is introduced to schedule $\tau$ with the help of the HCUFs. Finally, the algorithm's complexity and schedulability conditions are discussed.

### 8.3.1. Mapping $(m,k)$-Constraints to HCUFs

The HCUF introduced to model $(m,k)$-firm constraints is based on a TUF that rates the execution of single jobs. Each job $\tau_{i,j}$ of a task $\tau_i$ is classified by a step-shaped TUF $u_\mathrm{F}(\tau_{i,j})$ that represents a firm real-time requirement. $u_\mathrm{F}(\tau_{i,j})$ is evaluated when job $\tau_{i,j}$ finishes execution (at time $f_{i,j}$) or is cancelled. If $\tau_{i,j}$ is completed before its deadline, $u_\mathrm{F}$ yields a utility value of 1, and of 0 else:

$$u_\mathrm{F}(\tau_{i,j}) = \begin{cases} 1 & f_{i,j} \leq d_{i,j} \\ 0 & \text{else} \end{cases} \tag{8.9}$$

For each task $\tau_i$, a sliding *state window* $w_i = (w_i^1, \ldots, w_i^{k_i})$, $w_i^l \in \{0,1\}$ stores the utility values $u_\mathrm{F}(\tau_{i,j-k+l})$ of the last $k_i$ jobs, with $\tau_{i,j}$ being the most recent job that was finished and $w_i^{k_i}$ being its utility value. $w_i$ corresponds to the $k$-sequence $\sigma_i$ of a task (sect. 8.2.1), but uses a different indexing. A task's basic $(m,k)$-HCUF $H_\mathrm{m}(\tau_i)$ computes the mean value of all entries in $w_i$:

$$H_\mathrm{m}(\tau_i) = \frac{1}{k_i} \sum_{j=1}^{k_i} w_i^j \tag{8.10}$$

Obviously, if all jobs inside the window keep their deadlines, $H_\mathrm{m}(\tau_i) = 1$. The $(m,k)$-constraint of $\tau_i$ requires that $H_\mathrm{m}(\tau_i)$ never falls below $\frac{m_i}{k_i}$, i.e. $H_\mathrm{m}(\tau_i) \geq \frac{m_i}{k_i}$. If $H_\mathrm{m}(\tau_i) = \frac{m_i}{k_i}$, the task is in a critical state and its next job must not miss its deadline. Thus, the HCUF value representing the critical state is very important for scheduling decisions. However, these values are not directly comparable for tasks with different $(m,k)$-constraints. To overcome this drawback and ease scheduler implementation, $H_\mathrm{m}(\tau_i)$ is scaled by $\frac{k_i}{m_i}$ such that the minimally allowed utility for any $(m,k)$-constraint is 1:

$$\hat{H}_\mathrm{m}(\tau_i) = \frac{k_i}{m_i} H_\mathrm{m}(\tau_i) = \frac{1}{m_i} \sum_{j=1}^{k_i} w_i^j \tag{8.11}$$

$\hat{H}_\mathrm{m}$ gives a relative measurement of how far a task is away from a violation of its $(m,k)$-constraint. As long as $\hat{H}_\mathrm{m}(\tau_i) \geq 1$, the task's $(m,k)$-constraint is fulfiled. The maximum value of $\hat{H}_\mathrm{m}(\tau_i)$ is $\frac{k_i}{m_i}$, which leads to the following two properties of $\hat{H}_\mathrm{m}$ for special $(m,k)$-parameters:

**Property 8.1.** *Tasks $\tau_i, \tau_j$ that have different $(m,k)$-constraints, but whose constraints have the same $(m,k)$-ratio $\frac{m_i}{k_i} = \frac{m_j}{k_j}$, can gain the same maximum value of $\hat{H}_\mathrm{m}$, i.e. $\max \hat{H}_\mathrm{m}(\tau_i) = \hat{H}_\mathrm{m}(\tau_j)$. However, for tasks with bigger $k$, $\hat{H}_\mathrm{m}$ exhibits a more fine-grained resolution.*

**Property 8.2.** *Tasks $\tau_i, \tau_j, i \neq j$ with $k_i = k_j$ and $m_i < m_j$ have different maximum values $\max \hat{H}_\mathrm{m}(\tau_i) > \max \hat{H}_\mathrm{m}(\tau_j)$.*

The influence of these properties on scheduling decisions is discussed after the description of actual scheduling.

### 8.3.2. Scheduling

The scheduling algorithm MKU for $(m, k)$-tasks is based on LBESA (Jensen et al. 1985; Locke 1986). The MKU algorithm is formally shown in algorithm 8.1 which is executed any time a new job is released. All ready jobs are kept in a list $S$ ordered increasingly by their deadlines. Dispatching is performed in EDF manner from the front of $S$ (not shown in alg. 8.1). Newly arriving jobs are enqueued in EDF order into $S$. If, after the insertion of a new job into $S$, the possibility of a deadline miss is detected for a job $\tau_{i,j}$ (line 3), all jobs $\tau_{l,n}$ with deadlines $d_{l,n} \leq d_{i,j}$ (including $\tau_{i,j}$) are examined (line 4). For each job $\tau_{l,n}$, its task's *possible* HCUF $\hat{H}_{\mathrm{p}}(\tau_{l,n})$ value is calculated, assuming that $\tau_{l,n}$ is cancelled. $\hat{H}_{\mathrm{p}}(\tau_{l,n})$ is calculated in accordance with equation (8.11) as follows:

$$\hat{H}_{\mathrm{p}}(\tau_i) = \frac{1}{m} \sum_{j=2}^{k_i} w_i^j \tag{8.12}$$

Note that the sum now only ranges from $w_i^2$ up to $w_i^{k_i}$, compared to equation (8.11): If the current job is cancelled, $w_i^1$ will be removed from the window and a new value 0 representing that cancellation will be inserted at the end of $w_i$. The job with the maximum value of $\hat{H}_{\mathrm{p}}$ is then removed from the schedule. However, only jobs $j_{i,l}$ with $\hat{H}_{\mathrm{p}}(\tau_i) \geq 1$ are considered for cancellation (second condition in line 5). This procedure is repeated until the overload is resolved. If no candidates for removal can be found, obviously no job can be cancelled without violating its $(m, k)$-constraint, and therefore the whole task set is not schedulable under MKU with the given $(m, k)$-constraints.

**Algorithm 8.1:** The MKU algorithm

**Data**: $S$: Current schedule
**Input** : New job $\tau_N$ to enqueue into $S$
**Output**: Set $C$ of cancelled jobs

```
1 InsertInEDFOrder(S, τ_N);
2 C ← ∅;
3 while τ_{i,j} ←FindMissJob(S) do
4 |   P ← {τ^P ∈ S : d^P ≤ d_{l,n}} ;                    // Prospects for cancellation
5 |   τ^C ← τ ∈ P : Ĥ_p(τ^P) = min_{τ∈P} Ĥ_p(τ) ∧ Ĥ_p(τ^P) ≥ 1;
6 |   if τ^C does not exist then
7 |   |   ReportError();
8 |   RemoveJob(S, τ^C);
9 |   C ← C ∪ {τ^C};
```

For the following discussion of the properties from the previous section, it is assumed that all tasks' states $w_i$ are initialised with $1^{k_i}$. In the beginning, tasks with equal $\frac{m_i}{k_i}$ ratios have the same probability for having their first job cancelled. Due to the finer granularity of $\hat{H}_{\mathrm{m}}$ of tasks with bigger $k$ values (see property 8.1 from section 8.3.1), such tasks also have a higher probability to be chosen for subsequent cancellations, as their $\hat{H}_{\mathrm{m}}$ value decreases more slowly. For tasks $\tau_i, \tau_j$ with $k_i = k_j$, but $m_i < m_j$ (property 8.2 from section 8.3.1), $\tau_i$ has a higher probability of cancellation due to its higher $\max \hat{H}_{\mathrm{m}}(\tau_i)$ value, which represents $\tau_i$'s higher tolerance towards job losses.

### 8.3.3. Complexity

The complexity of the MKU approach can be split in two parts: Concerning regular management of the ready list, MKU inherits the complexity of EDF. A naive implementation using regular lists has a complexity of $O(n)$ for insertion of new jobs, where $n$ is the size of the task set. Using balanced trees, the management complexity can be reduced to $O(\log n)$.Then however, additional overhead can be expected to keep the tree balanced. For the purpose of this work the use of regular lists is assumed.

Additionally, MKU introduces some overhead for detection and resolution of overload situations (lines 3 to 9 of alg. 8.1). These operations must only be performed when a new job is added to the ready list. Overload detection (line 3), i.e. finding a job that will miss its deadline, takes $O(n)$ steps, as the ready list must be examined from its beginning. Finding a job for cancellation (lines 4 and 5) requires another walk through the list. The $\hat{H}_\mathrm{m}$ value for each task should be initialised once and then be kept in memory. Thus, online calculation/update of $\hat{H}_\mathrm{m}$ and $\hat{H}_\mathrm{p}$ values can be performed in constant time (regardless of the window size $k$) by advancing the existing value using the entries of the sliding window $w_i$ of each task $\tau_i$. Removing the job in line 8 can be performed with constant overhead, if an appropriate list structure (double-linked list) is used. The loop is executed for at most $n$ times, leading to a complexity of $O(n^2)$ for the MKU algorithm.

### 8.3.4. Schedulability

The schedulability test devised by Goossens (J. Goossens 2008) for task sets under DBP scheduling also applies to the MKU scheduler in terms of the upper bound for simulation. The test is solely based on the tasks' periods and $(m,k)$-constraints, and the fact that the DBP scheduler itself is memoryless. The MKU scheduler itself does not possess an internal state. Like DBP, it acts solely on the states of the tasks, namely their $k$-sequences to calculate a task's HCUF.

When applying GST for MKU scheduling, a further slight optimisation is possible. GST examines the system state $\sigma = (\sigma_1, \ldots, \sigma_n)$ after each hyperperiod. A repetition of $\sigma$ without any task violating its $(m,k)$-constraint means that the schedule is cyclic and valid, as the schedule itself solely depends on $\sigma$. In its calculation of the possible HCUF $\hat{H}_\mathrm{p}$ (eq. (8.12)), the MKU scheduler only regards the most recent $k_i - 1$ entries of each $\sigma_i$. Insofar, it operates on a *reduced system state*:

**Definition 8.1.** *Let $\sigma^R = (\sigma_1^R, \ldots, \sigma_n^R)$ be the* reduced system state *of a set of $(m,k)$-firm real-time tasks. $\sigma_i^R$ is obtained from system state $\sigma_i = (\sigma_i^{j-k+1}, \ldots, \sigma_i^{j-1}, \sigma_i^j)$ by ignoring the least recent entry, i.e. $\sigma_i^R = (\sigma_i^{j-k+2}, \ldots, \sigma_i^{j-1}, \sigma_i^j)$.*

The following theorem provides the basis for an optimisation of GST for the HCUF-based scheduler:

**Theorem 8.3.** *If during the execution of GST with MKU a reduced system state $\sigma^R$ recurs at a hyperperiod boundary, a cycle in the MKU schedule has been found.*

*Proof.* Let $\sigma^1, \sigma^2$ be two system states incurred in this order during execution of GST with MKU, such that for the derived system states $\sigma^{R,1} = \sigma^{R,2}$ (in the following simply $\sigma^R$. Further, let $L(\sigma) = (\sigma_1^{j-k+1}, \sigma_2^{j-k+1}, \ldots, \sigma_n^{j-k+1})$ be the vector of a system state $\sigma$'s least recent entries that are ignored by $\sigma^R$. Two cases can be distinguished:

1. If $L(\sigma^1) = L(\sigma^2)$, then also $\sigma^1 = \sigma^2$. The whole system state recurred, a cycle in the schedule has been found.
2. If $L(\sigma^1) \neq L(\sigma^2)$, then there exists at least one $i$ such that $\sigma_i^{1,j-k_i+1} \neq \sigma_i^{2,j-k_i+1}$. However, the schedule $S_1$ produced by MKU between $\sigma^1$ and $\sigma^2$ solely depends on $\sigma^{R,1}$, as MKU regards only the $\sigma_i^R$ for its cancellation decisions. Thus, MKU will produce the same schedule $S_2 = S_1$ after $\sigma^2$, as $\sigma^{R,1} = \sigma^{R,2}$.

□

In case 1, a real cycle has been found, as is also detected by GST. Case 2 needs some closer inspection, as it can help to speed up the schedulability test:

**Corollary 8.1.** *Let $\sigma^1, \sigma^2$ be two system states such with $\sigma^{R,1} = \sigma^{R,2}$, $L(\sigma^1) \neq L(\sigma^2)$ (case 2 in the proof of theorem 8.3), $\sigma^R := \sigma^{R,1} (= \sigma^{R,2})$, and $\sigma^2$ occurs after $\sigma^1$. Further, let $\sigma^3$ be the next system state with $\sigma^{R,3} = \sigma^R$. Then, $L(\sigma^3) = L(\sigma^2)$ and $\sigma^3 = \sigma^2$.*

*Proof.* Recall that the schedule $S_1$ produced by MKU after $\sigma^1$ only depends on $\sigma^{R,1} = \sigma^R$. Thus, MKU will produce the same schedule $S_2 = S_1$ after $\sigma^2$. As the decisions for $S_1$ and the result $\sigma^{R,2}$ are solely based on $\sigma^{R,1}$, $S_2$ (taking the same decisions) will produce the same result $\sigma^3 = \sigma^2$, and thus $L(\sigma^3) = L(\sigma^2)$. □

This means that the schedulability test for MKU **may** terminate at least one hyperperiod earlier compared to GST, depending on the length of the cycle.

## 8.4. Evaluation Methodology

An evaluation of the $(m, k)$-schedulers is based on extensive simulations of randomly generated task sets. The simulations are conducted using the `tms-sim` framework (Kluge 2014). This section presents the methodology that underlies the experimental evaluations.

### 8.4.1. Abstract and Concrete Task Sets

Abstract task sets form the basis of the simulations. An ATS $\alpha = \{\alpha_1, \alpha_2, \ldots, \alpha_n\}$ is a set of abstract tasks $\alpha_i = (e_i, T_i, m_i, k_i)$ with periods $T_i$, $(m, k)$-constraints $(m_i, k_i)$ and an execution time weights $e_i$. A concrete task set $\tau(\alpha, U_\mathrm{T})$ is derived from an ATS $\alpha$ by calculating the tasks' execution times $C_i$ such that the CTS approximates a target utilisation of $U_\mathrm{T}$. The execution time weight $e_i$ specifies, how much task $\tau_i$ contributes to the task set utilisation:

$$\frac{e_i}{\sum_{j=1}^{n} e_j} = \frac{U_i}{U_T} \tag{8.13}$$

Thereby, $U_i = \frac{C_i'}{T_i}$ is the utilisation of the task under consideration. Solving with eq. (8.13) for $C_i'$ yields:

$$C_i' = \frac{U_\mathrm{T}}{\sum_{j=1}^{n} e_j} T_i e_i \tag{8.14}$$

As only integral execution times are considered, $C_i$ is obtained from $C_i'$ through rounding:

$$C_i = \begin{cases} [C_i'], \text{ if } [C_i'] > 0 \\ 1, \text{ else} \end{cases} \tag{8.15}$$

Thereby, the operation $[x]$ stands for regular rounding, i.e. returns the integer value that is nearest to $x$. Through the rounding, the task set's actual utilisation $U = \sum_{i=0}^{n} \frac{C_i}{T_i}$ can deviate from the target utilisation. Task set generation is configured such that generated ATSs that deviate more than a constant $d_U$ from an initial target utilisation are automatically discarded.

### 8.4.2. Task Parameters

The parameters of the ATSs are generated using the `libc` pseudo-random number generator (`rand_r()`). Execution time weights are chosen from an interval $[1, e_{\max}]$ where $e_{\max}$ represents the granularity of the weights.

For the task periods $T_i$, two approaches are implemented: Either, the periods are chosen randomly from a given interval $[T_{\min}, T_{\max}]$. Alternatively, the period generator from Goossens and Macq (J. Goossens and Macq 2001) can be used. This generator yields task periods that have many common divisors, and a limited hyperperiod compared to randomly chosen periods. Period generation is based on a matrix of multipliers, where each row contains powers of a prime number. Period generation randomly selects one entry from each row. The actual period then is the product of the chosen entries. For the simulations, an additional restriction ensures that periods are $> 2$ for any task.

The $k_i$ parameters are chosen from an interval $[k_{\min}, k_{\max}]$. For the $m_i$ parameters, again two approaches are implemented: Either, they can be chosen from $[1, k_i]$. This approach can yield very low $m_i$ values (compared to $k_i$) that may seem quite unrealistic. Therefore, $m_i$ can be limited to meaningful ranges. An additional parameter $r_{\mathrm{m}} \in [0, 1]$ can be specified to lower-bound the $m_i$. The actual $m_i$ parameter then is chosen from $[r_{\mathrm{m}} k_i, k_i]$.

So far, there is not yet an efficient way to find good initialisations for the $k$-sequences. Checking all possible initialisation values is not feasible, as $2^{\sum_{i=1}^{n} k_i}$ schedules would have to be checked. Therefore, $1^{k_i}$ is used as initial $k$-sequence, which might be as good or bad as any other (possibly random) choice.

### 8.4.3. Simulation

Simulations can be performed for a given limited number of steps. This approach may yield false positives, i.e. even though a task set is executed successfully during the specified number of steps, it may still fail a later point in time. Nevertheless, this approach gives a first impression of the performance of the different scheduler. Also, it can easily be applied to all schedulers.

Alternatively, simulations can be performed using the exact schedulability tests. For MKP and MKP-S, the methods described by Jia et al. (Jia et al. 2007) (sufficient condition) and the one introduced in section 8.2.4 are used. For all other schedulers, GST (J. Goossens 2008) is applied. Simulation of a CTS is performed until a definite result concerning its feasibility is reached. This simulation approach is used to search for the breakdown utilisations (Lehoczky et al. 1989) of ATSs under different schedulers. The breakdown search works as follows: A single ATS is repeatedly used to generate CTSs. The first ATS is generated using a target utilisation $U_{\mathrm{T}} = U_{\mathrm{B}}$. If the CTS is found to be schedulable for a certain scheduler, $U_{\mathrm{T}}$ is increased by a utilisation step $s_{\mathrm{U}}$. Using this updated $U_{\mathrm{T}}$, a new CTS is derived from the ATS and another simulation is performed. This process is repeated until the CTS is no longer schedulable. The last $U_{\mathrm{T}}$ that yields a feasible CTS is called the *breakdown utilisation*.

To account for breakdown anomalies (see sect. 8.2.4), $U_{\mathrm{T}}$ is increased further and the derived CTSs are simulated, too. This process stops when the derived CTSs no longer fulfil the necessary schedulability condition $U_{\mathrm{mk}} \leq 1$ (see sect. 8.2.4).

An overview of the task models and schedulers used for evaluation can be found in table 8.2. For simulations, the gMUA-MK approach is adjusted to immediately cancel jobs that are removed from a schedule due an overload. Due to the assumption of constant execution times, they would be cancelled anyway. Like in the MKU approach, the TUF for firm real-time tasks (see eq. (8.9)) is used for calculation of value-density.

Table 8.2.: Task models, schedulers, and schedulability tests used in the experimental evaluation

| Model | Abbr. | Reference | Test |
|---|---|---|---|
| *FPP Scheduler* | | | |
| Distance-based priority | DBP | Hamdaoui and Ramanathan 1995 | GST |
| Fixed $(m, k)$-patterns | MKP | Ramanathan 1999 | Jia et al. 2007, sect. 8.2.4 |
| MKP with pattern rotation | MKP-R | Quan and Hu 2000 | sect. 8.2.4 |
| *EDF-based Schedulers* | | | |
| Guaranteed Dynamic Priority Assignment | GDPA | Cho, Chung, et al. 2010 | GST |
| Simplified GDPA | GDPA-S | Cho, Chung, et al. 2010 | GST |
| Global Multiprocessor Utility Accrual scheduling for (m, k)-firm deadline-constraints | gMUA-MK | Rhu et al. 2011 | GST |
| Utility-based $(m, k)$-tasks | MKU | sect. 8.3 | GST |

### 8.4.4. Parameters & Aims of the Evaluation

#### Parameters

An overview of the parameters used for task set generation and simulation can be found in table 8.3. The are passed to `tms-sim` via the command line or a parameter file. The actual values assigned to these parameters for simulations are presented in section 8.5 together with the corresponding results.

#### Aims

The experimental evaluations aim to answer the following questions:

1. Initial results (Kluge et al. 2015) that are subject to consolidation (sect. 8.5.1) indicate remarkably performance differences between the different schedulers, when using the ratio of task sets that are feasible as a performance metric. How do the different schedulers compare against each other, when an exact schedulability test is applied?

Table 8.3.: Parameters for task set generation and simulation

| Symbol | Description |
|---|---|
| $[T_{\min}, T_{\max}]$ | Range for task periods (ignored when Goossens' and Macq's period generator (J. Goossens and Macq 2001) is used) |
| $e_{\max}$ | Granularity of execution time weights |
| $[k_{\min}, k_{\max}]$ | Range for the $k_i$ parameter |
| $[m_{\min}, m_{\max}]$ | Range for $m_i$ paramter, usually $[2, k_i]$ |
| $r_{\mathrm{m}}$ | If specified, restricts $m_i$ to $[r_{\mathrm{m}} k_i, k_i]$ |
| $U_{\mathrm{T}}$ | Target/base utilisation of generated task sets |
| $d_{\mathrm{U}}$ | Maximum allowed deviation from $U_{\mathrm{T}}$ |
| $s_{\mathrm{U}}$ | Utilisation step for breakdown utilisation search |

2. How pessimistic is Goossens' feasibility interval (eq. (8.3), J. Goossens 2008)? Even for small task periods and $m_i$ resp. $k_i$ values, the interval can get quite large. How high are the savings in terms of simulated time introduced through GST? Also, the practical relevance of the optimised schedulability test for MKU that can be derived from theorem 8.3 resp. corollary 8.1 must be examined.

3. How relevant are breakdown anomalies (see sect. 8.2.4)?

4. As Goossens (J. Goossens 2008) shows, the initialisation of the $k$-sequences of tasks can impact the feasibility of a task set. However, it is open how to find good initialisation values. The possibility of cross-initialisation of $k$-sequences between different schedulers is explored: If a task set with given initial $k$-sequence is feasible only under one of two schedulers, simulation of the successful scheduler necessarily runs into a circle of $k$-sequences when applying GST. Is it possible to use one of the recurring $k$-sequences as initial value for execution with the hitherto failing scheduler?

5. Due to cancellations, processing time already spent by the cancelled jobs is lost. How much performance is lost by the different schedulers?

6. In general, the evaluations are based on arbitrary task sets with random parameters which may not always have practical counterparts. How are the results for the above questions influenced, if task periods and/or $m$ parameters are restricted to realistic ranges/values?

## 8.5. Results

In the following, the results of experimental evaluations are presented. First, the results from (Kluge et al. 2015) are consolidated. Then, results from two groups of simulations are described. In the first group (sect. 8.5.2), arbitrary task sets are examined in order to answer the first five questions laid down in section 8.4.4. The second group (sect. 8.5.3) deals with the use of realistic task parameters (last question in sect. 8.4.4). These are only examined from the performance point of view.

### 8.5.1. Comparison of All Approaches

In (Kluge et al. 2015), initial results on the performance of some of the schedulers listed in table 8.2 were presented. These results are based on the simulation of random task sets for a fixed number of time steps. If during this simulation time no violation of an $(m, k)$-constraint is detected, the task set is classified as feasible. This approach can yield false positive results, as an infeasibility may also happen just after the given number of time steps. Nevertheless, these results gave a first impression of the performance of the schedulers: Best results were achieved with the DBP and MKU approaches, followed by MKP and MKP-S. Least performance was exhibited by GDPA. Due to a bug in the implementation of the simulator, approaches based on FPP scheduling (DBP, MKP, MKP-S) exhibited a lower performance than they actually have. Also, MKU showed better performance than DBP for moderate overloads.

The aim of the following experiment is a rough comparison of all approaches listed in table 8.2 and the consolidation of the results reported in (Kluge et al. 2015). The parameters used for task set generation can be found in table 8.4. Each task set is executed for 1,000,000 time steps.

Table 8.4.: Task set parameters for comparison of $(m, k)$-schedulers

| Parameter | Value |
|---|---|
| $p_{\min}$ | 5 |
| $p_{\max}$ | 15 |
| $k_{\min}$ | 2 |
| $k_{\max}$ | 10 |
| $m_i$ | $[2, k_i]$ |
| $d_U$ | 0.05 |
| $U_T$ | $\{1.05, 1.15, \ldots, 1.95\}$ |

Figure 8.2 shows the success rates that are achieved by the different scheduling approaches. Comparing the fixed-patter-based approaches MKP and MKP-S, the improvements that are introduced by the rotation of the $(m, k)$-patterns in MKP-S are obvious. The EDF-based approaches GDPA/GDPA-S and GMUA-MK all show a very low performance: Even for a moderate overload of $U_T = 1.05$ they stay up to 50 % behind the other approaches. With increasing utilisation, their performance decreases radically. Already at a target utilisation of $U_T = 1.35$, less than 3 % of the task sets are feasible with these approaches. Similar performance setbacks can be observed for the MKP/MKP-S approaches. The performance number for these approaches confirm the results by Quan and Hu (Quan and Hu 2000). Best results are achieved with the DBP and MKU approaches. Thereby, DBP achieves a slightly higher performance than MKU for low overloads and improves with increasing target utilisation.

It is important to keep in mind that actual performance numbers may be lower when applying an exact schedulability test. Nevertheless, the results show that best performance can be achieved with DBP and MKU.

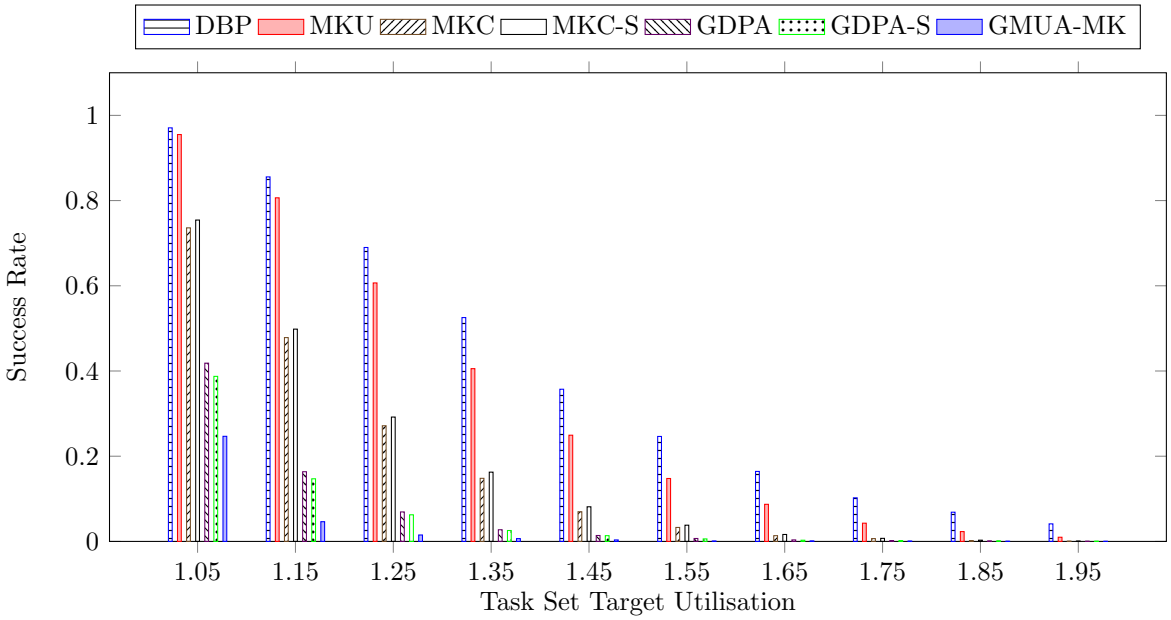



Figure 8.2.: Performance of $(m, k)$-schedulers (see tab. 8.2)

### 8.5.2. Arbitrary Task Sets and Exact Schedulability Test

As already stated above, the simulation of a task set for a fixed number of time steps can yield false positive results. For a closer examination of the different approaches, the exact schedulability tests (see tab. 8.2) are applied. This experiment is combined with a search for the breakdown utilisation of an ATS (see sect. 8.4.3). ATSs are executed beyond this point to account for breakdown anomalies (sect. 8.2.4). Parameters for generation of ATSs used in this experiment are shown in table 8.5. CTSs derived from these ATSs are simulated as deemed necessary by the schedulability tests. Two simulations are examined: Simulation 1 (S1 in the following) consists of 500 ATSs and is executed with all schedulers in table 8.2. Simulation 2 (S2) consists of 10,000 ATSs and is executed only with the DBP and MKU schedulers. Beyond performance ratings of the scheduling approaches, also the performance of GST, breakdown anomalies, and cross-initialisation of $k$-sequences between schedulers are investigated. If not mentioned explicitly, breakdown anomalies are ignored in the following results.

#### Scheduler Performance

The overall performance of all schedulers in S1 is shown in figure 8.3. The bars stand for the ratio of ATSs that are schedulable at least up to the indicated target utilisation (not accounting for breakdown anomalies). Comparing these numbers with the results of the rough estimation in sect. 8.5.1 (fig. 8.2) shows that the actual success ratio of all schedulers is lower, which is just to be expected due to false positives. Nevertheless, the ratios between the different schedulers remain nearly unchanged. So, the rough estimation at least allows for qualitative comparison of the schedulers.

Table 8.5.: Task set parameters for exact schedulability test

| Parameter | Value |
|---|---|
| $p_{\min}$ | 5 |
| $p_{\max}$ | 60 |
| $k_{\min}$ | 2 |
| $k_{\max}$ | 10 |
| $m_i$ | $[2, k]$ |
| $d_{\mathrm{U}}$ | 0.05 |
| $U_{\mathrm{B}}$ | 1.05 |
| $s_{\mathrm{U}}$ | 0.1 |

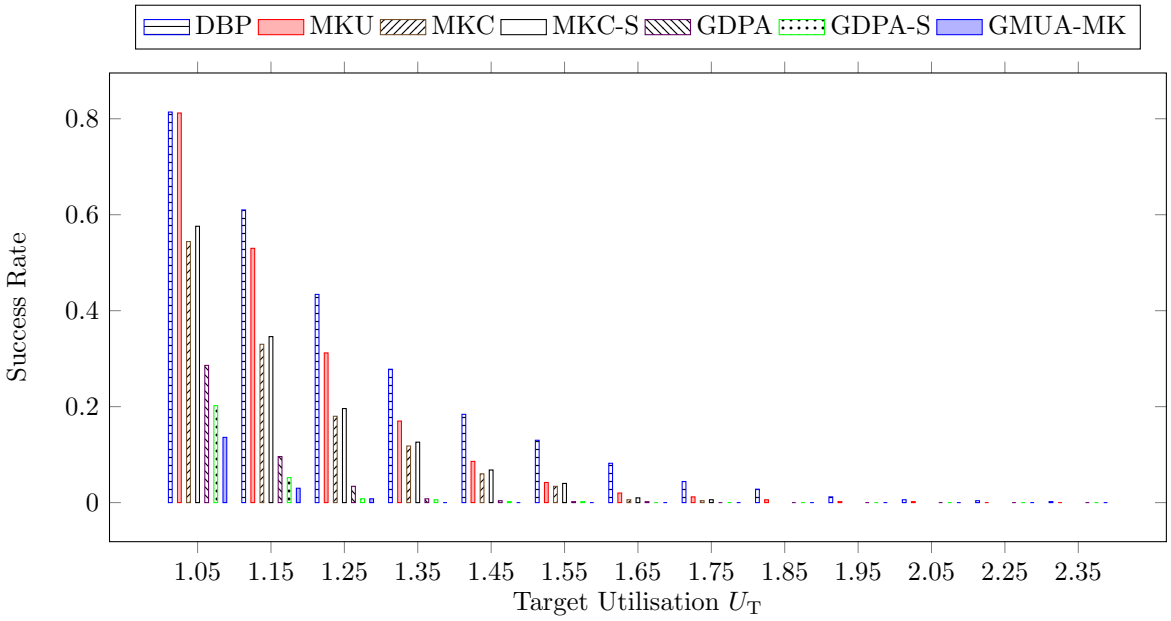



Figure 8.3.: Ratio of ATSs in S1 that are feasible up to a certain target utilisation $U_{\mathrm{T}}$

The overall performance in S2 is shown in figure 8.4. While for a target utilisation $U_{\mathrm{T}} = 1.05$ both DBP and MKU yield similar performance, for higher target utilisations the superiority of DBP becomes obvious. In this simulations, 25,931 task sets are feasible under DBP and 20,185 ones under MKU. Again, these numbers do not account for breakdown anomalies. In this simulation, a higher target utilisation is reached than in S1. This can be explained through the higher number of task sets that are examined, and thus the greater variability of task parameters.

Beyond providing a necessary schedulability condition, the $(m, k)$-utilisation $U_{\mathrm{mk}}$ does not help further to estimate the feasibility of a task set. Figure 8.5 shows the occurrence rate of the $(m, k)$-utilisations at the breakdown point of a task set, classified by rounding to the nearest tenth. The class $U_{\mathrm{mk}} = 0$ stands for ATSs that are not feasible at all. Obviously, a high $(m, k)$-utilisation does not per se prohibit feasibility, although this is achieved by only

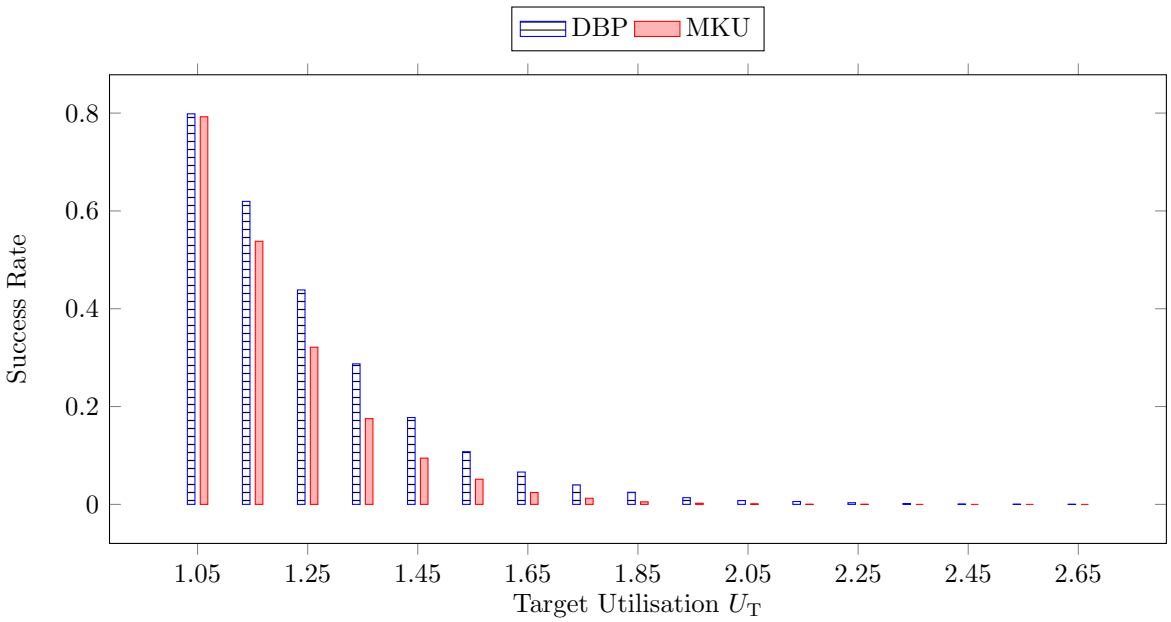



Figure 8.4.: Ratio of ATSs in S2 that are feasible up to a certain target utilisation $U_T$

few ATSs. Most ATSs have a $(m, k)$-utilisation in the interval $[0.65, 0.85)$. The histogram for S2 is similar and is therefor omitted.

Figure 8.6 shows a scatterplot of the $(m, k)$-utilisations of ATSs at their breakdown points in S2. The $(0, 0)$ coordinate contains ATSs that are not schedulable at all according to the breakdown metric. Points lying near the axes (having one coordinate 0) represent ATSs that are schedulable with only one of the approaches. As already shown in the above histogram, for increasing utilisation (and thus increasing $U_{mk}$), fewer ATSs are feasible. Most ATSs are located inside a cloud around the diagonal of the plot. While having a low $(m, k)$-utilisation does not guarantee feasibility, the diagram shows a clear trend: If a feasible CTS can be derived from an ATS for one of the DBP or MKU schedulers, then feasibility beyond $U_{mk=0.4}$ can be expected in most cases.

### Performance of Schedulability Tests

The exact schedulability test for the approaches based on fixed $(m, k)$-patterns proposed in section 8.2.4 yields a large feasibility interval. However, simulations must only be performed if the sufficient test (Jia et al. 2007) fails. In contrast, the feasibility interval for DBP (J. Goossens 2008) tends to exceed MKP feasibility interval by far. In the following, the focus is on the gains that are obtained through GST.

The DBP feasibility interval defines a very high bound for the number of hyperperiods that must be simulated successfully until feasibility of a task set can safely be assumed. The experiments show that this bound is quite pessimistic and the optimisation incorporated in GST yields great value for the schedulability test. S1 contains 500 infeasibility CTSs (those after the breakdown point). In all schedulers using GST, infeasibility is detected for most task sets ($> 99\,\%$) during the first hyperperiod, only few take longer. The longest observed

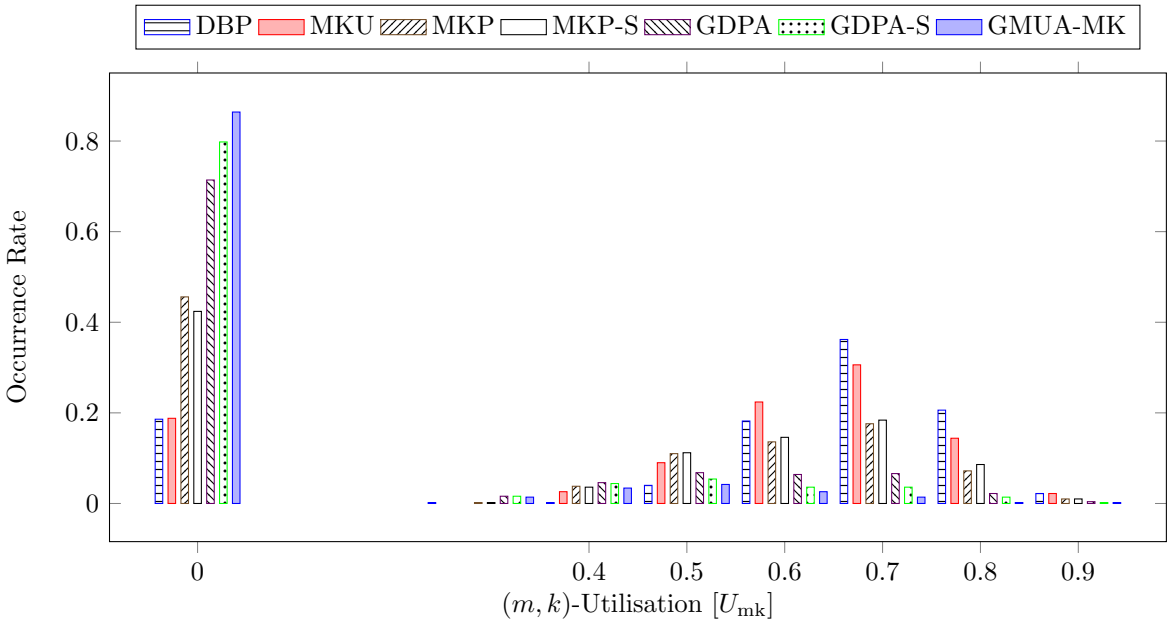



Figure 8.5.: Breakdown $(m, k)$-utilisations of S1 (classified by rounding to nearest tenth)

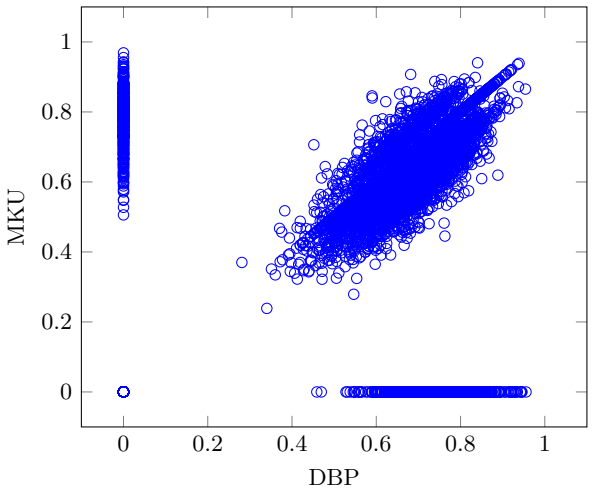



Figure 8.6.: $(m, k)$-utilisations at breakdown point for DBP and MKU; each point stands for an ATS

time is six hyperperiods using the DBP scheduler. Feasibility is mostly found after the second hyperperiod, again with only few CTSs needing more (up do 13 hyperperiods can be observed). For feasible CTSs, the first hyperperiod can be seen as a warm-up phase: At the start, the $k$-sequences have an arbitrary initialisation, in our case $1^k$. These $k$-sequences are very unlikely to recur, as at least some jobs necessarily must be cancelled due to the overload. So, during the warm-up phase a good initialisation for the $k$-sequences is found, which leads into a recurring system state.

The results in S2 are similar. There, 10,000 CTSs are infeasible for each of the DBP and MKU approaches. For DBP, infeasibility is detected for 9,830 (98 %) task sets during simulation of the first hyperperiod, and for 124 task sets during the second hyperperiod. Only few task sets take longer – the longest observed simulation finishes during the 13th hyperperiod. The number for MKU are similar: For 9,901 (99 %) task sets the simulation detects infeasibility during the first hyperperiod, and for 79 task sets during the second hyperperiod. The remaining task sets take longer, but also here infeasibility is detected latest during the 13th hyperperiod. Detection of feasibility usually takes slightly longer, again. For DBP, a total of 25,931 task sets is found to be feasible. Only 68 simulations finish after the first hyperperiod. Most task sets ($24,761 \approx 95\,\%$) are found to be feasible after the second hyperperiod. Decreasing numbers of task sets take longer – the longest observed simulation runs for 21 hyperperiods. Again, the behaviour of MKU is similar: In total, 20,185 task sets are schedulable under MKU. From these, 105 finish successfully after the first hyperperiod. The largest amount ($19,598 \approx 97\,\%$) is again found to be feasible after the second hyperperiod. The remaining task sets are distributed with decreasing number over longer time spans up to 21 hyperperiods.

The gain from the optimised schedulability test for MKU that follows from theorem 8.3 and corollary 8.1 is only marginal. From 997 CTSs in S1 that are feasible under MKU, only 2 (0.2 %) would finish earlier. In S2, from 20,185 CTSs schedulable under MKU, 57 ($< 0.2\,\%$) could profit from the optimised test.

#### Breakdown Anomalies

Extending the simulations described previously beyond the ATSs' breakdown points yields the following results. The numbers for S1 are shown in table 8.6. Recall that for MKP and MKP-S no such anomalies can occur (theorem 8.2 in sect. 8.2.4). For all other schedulers, no resp. only few such anomalies are found. From the 10,000 ATSs in S2, 55 exhibit a breakdown anomaly under DBP scheduling. Under MKU scheduling, 93 task sets are affected by this behaviour. For 5 ATSs, a breakdown anomaly is detected both under DBP and MKU.

Table 8.6.: Incidence of breakdown anomalies in S1, based on 500 ATSs

| DBP | MKU | GDPA | GDPA-S | GMUA-MK |
|---|---|---|---|---|
| 2 | 4 | 0 | 2 | 6 |

#### Cross-Initialisation of $k$-Sequences

As noted by Goossens (J. Goossens 2008), the choice of the initial $k$-sequence can have significant impact on the feasibility of a task set. So far, no efficient algorithm is available

that can derive a meaningful initialisation. In the simulations S1 and S2, there exist CTS that are feasible only under one of any two approaches, but not under both. Feasibility in this case means that, after an initial warm-up phase which started with each task's $k$-sequence $\sigma_i$ being initialised to $1^{k_i}$, a system state $\sigma = (\sigma_1, \sigma_2, \ldots, \sigma_n)$ (see sect. 8.2.1) periodically recurs. For a CTS that is feasible under two scheduling approaches, the corresponding system states may be different.

The idea of this experiment is to use a periodically recurring system state of a CTS that is only feasible under one of two schedulers for initialisation of the same CTS under the other scheduler. This approach might especially be interesting to improve the performance of approaches like GDPA. More formally, assume a CTS $\tau$ with $\sigma_i = 1^{k_i}, i = 1, 2, \ldots, n$ that is feasible under a scheduler $SC_\mathrm{F} \in \mathit{Schedulers} = \{\mathrm{DBP}, \mathrm{MKU}, \mathrm{GDPA}, \mathrm{GDPA\text{-}S}, \mathrm{GMUA\text{-}MK}\}$, but not under another scheduler $SC_\mathrm{I} \in \mathit{Schedulers}$. Thus, the execution of $\tau$ using $SC_\mathrm{F}$ finally runs into a cycle where a system state $\sigma^\mathrm{F}$ with $\sigma_i^\mathrm{F} \neq 1^{k_i}$ recurs periodically. This follows from the fact that the task set is overloaded and thus some jobs must be cancelled. Now, derive a task set $\tau'$ from $\tau$ by initialising each task $\tau_i'$'s $k$-sequence with the corresponding data from $\sigma^F$. $\tau'$ then is simulated under $SC_\mathrm{I}$ using GST to check whether the new initial $k$-sequence leads to a valid schedule. This approach is not applicable to the MKP and MKP-S approaches, as these disregard tasks' $k$-sequences.

Table 8.7 show the results of applying the idea of cross-initialisation to the results from S1. For each transfer, the number of task sets that are candidates for the transfer, and the percentage of tasks sets where this transfer yields a feasible schedule are shown. $k$-sequences are transferred from rows to columns. For example, there are 382 task sets that are feasible under DBP, but not under MKU when using $1^k$ as initial $k$-sequence. When using the final $k$-sequence from a successful DBP schedule, 0.8 % of these are feasible under MKU. For most transfers, the gain is $\leq 1$ % or does not exist at all. The only transfers where noticeable results can be achieved are those from MKU to DBP (3.1 %) and from GDPA to GDPA-S (7.1 %).

Table 8.7.: Number of relevant task sets and success rates (%) for cross-initialisation of $k$-sequences

| From – To | DBP | | MKU | | GDPA | | GDPA-S | | GMUA-MK | |
|---|---|---|---|---|---|---|---|---|---|---|
| DBP | - | - | 382 | 0.8 | 1068 | 0.0 | 1137 | 0.2 | 1178 | 0.5 |
| MKU | 65 | 3.1 | - | - | 348 | 0.0 | 379 | 0.3 | 396 | 1.0 |
| GDPA | 5 | 0.0 | 9 | 0.0 | - | - | 14 | 7.1 | 25 | 0.0 |
| GDPA-S | 3 | 0.0 | 1 | 0.0 | 0 | 0.0 | - | - | 10 | 0.0 |
| GMUA-MK | 2 | 0.0 | 0 | 0.0 | 4 | 0.0 | 4 | 0.0 | - | - |

Using the results from S2 yields similar numbers. In S2, 1,474 task sets are feasible only under MKU. Using cross-initialisation, another 30 of these ($\approx 2.0$ %) are made feasible under DBP. In the other direction, from 7,220 task sets feasible only under DBP another 59 ($\approx 1.0$ %) are made feasible under MKU. Thus, cross-initialisation can improve schedulability only marginally.

### Cancellation of Running Jobs

Cancelling a job that has already started execution leads to the already consumed processing time being lost. Figure 8.7 shows the mean ratio of processing time that is lost due to cancellation of executing jobs in S1, figure 8.8 shows the same numbers for S2. The schedulers based on fixed $(m, k)$-patterns (MKP, MKP-S) are omitted for two reasons: (1) If the sufficient schedulability test is successful, no simulation is performed, so no numbers are available for some task sets. (2) Only optional jobs (having lowest possible priority) are allowed to be cancelled; processing time that would be lost could be reclaimed by (possibly non-real-time) tasks that are running above the lowest possible priority, but still below the priorities of the $(m, k)$-firm real-time tasks.

Only CTSs that are feasible under all scheduler are included in the figures. The numbers are calculated in the following manner: For each CTS, the number of lost time steps is scaled by the task set's hyperperiod and the number of hyperperiods it is executed. From these numbers, the average is calculated for each target utilisation.

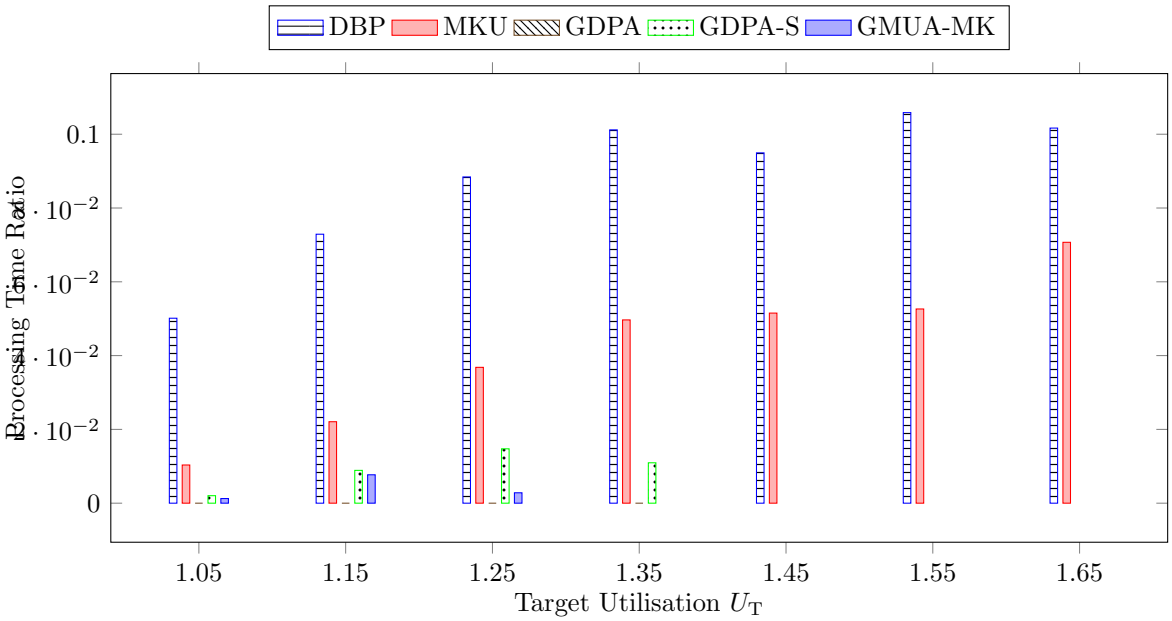



Figure 8.7.: Mean lost processing time through EC in S1 (only feasible task sets; scaled by number of hyperperiods and hyperperiod length); only task sets where all schedulers successful

Figure 8.7 shows that approaches that show a rather low performance (in terms of their ability to find finding feasible schedules), tend to loose only a minor portion of processing time due to cancellation of already executing jobs. The losses in the GDPA, GDPA-S and GMUA-MK approaches stay well below 2 %. Most interestingly, in the GDPA no processing time is lost at all. This can be explained through the special technique in which GDPA calculates its schedule. Jobs with a high distance from dynamic failure are later considered for insertion into the EDF schedule as those that are near to a dynamic failure. If the insertion makes the EDF schedule infeasible, the job is removed again and deferred (but not yet cancelled!). Jobs

with high distance to dynamic failure tend to be considered rather late for the schedule, and thus have a higher probability to lead to infeasibility, as the schedule might already be rather "full" through more critical jobs. Thus they are deferred without being executed, until they are cancelled due to missing their deadline.

The results allow also to deduct that the higher flexibility of the DBP and MKU approaches (in terms of finding feasible schedules) is bought at the cost of a higher amount of lost processing time (here up to 10 %). Thereby, lower costs are incurred by MKU. This is due to the fact that MKU cancels jobs in a more anticipatory manner as soon as an overload pends somewhere in the scheduler. Jobs in DBP are only cancelled when they can no longer meet their deadline. Figure 8.8 gives a similar view on this issue. While MKU looses less processing time for moderate overloads, both approaches converge with increasing overloads. In very high overloads (beyond $U = 2.0$), MKU looses even more processing time than DBP. However, it is important to note that with increasing utilisation the pool of data is radically diminishing. While the numbers for $U_\mathrm{T} = 1.05$ are based on 7,385 task sets, both for $U_\mathrm{T} = 2.15$ and $U_\mathrm{T} = 2.25$ only one task set is available.

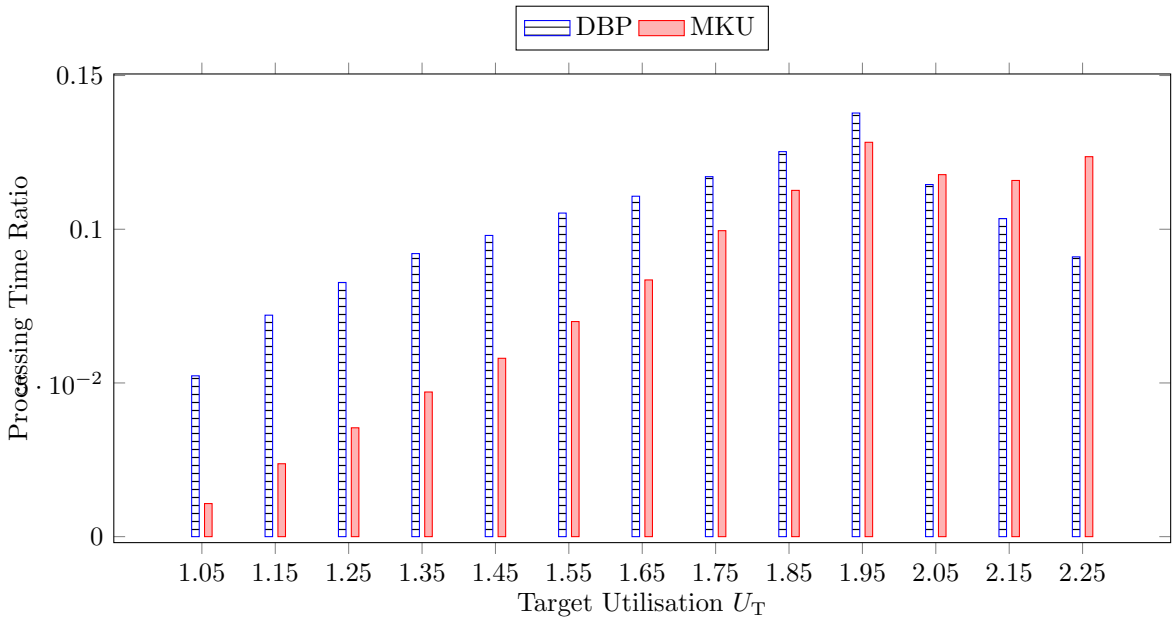



Figure 8.8.: Mean lost processing time through EC in S2 (only feasible task sets; scaled by number of hyperperiods and hyperperiod length); only task sets where both schedulers successful.

### 8.5.3. Realistic Periods and $m$ Parameters

The results presented so far are based on task sets with quite arbitrary parameters, concerning especially the task periods and $m$ parameters. In reality, one would not find such a great variability: In real applications, task periods within a task set can be tuned to be harmonic or at least have many common divisors (see e.g. Kramer et al. 2015). Usually, the periods span several orders of magnitude. Also, it seems unrealistic to have tasks with a low ratio $\frac{m}{k}$ which

would mean that most jobs could be skipped. In the following, the behaviour of DBP and MKU is examined under more realistic conditions by restricting $m$ parameters, task periods, and both. All other task set parameters in the following simulations are the same as in the previous section (see table 8.5).

### Restricted $m$ Parameters

Restricting the $m$ parameter of a task during task set generation to an interval $[r_m k_i, k_i]$, yields some interesting results. In the following evaluations, values $r_m \in \{0.1, 0.2, \ldots, 0.9\}$ are examined. Although reasonable $r_m$ values would rather be in the upper part of this set, it is also necessary to look at low values, as the following results will show.

For each value of $r_m$, 500 ATSs are generated and simulated again with all schedulers. Like before, the breakdown utilisation of the ATSs and the number of feasible CTSs for each target utilisation is used to assess the performance of the schedulers. Breakdown anomalies occur in these simulations only rarely ($\leq 2\,\%$ of the ATSs per $r_m$ value), so they can safely be ignored.

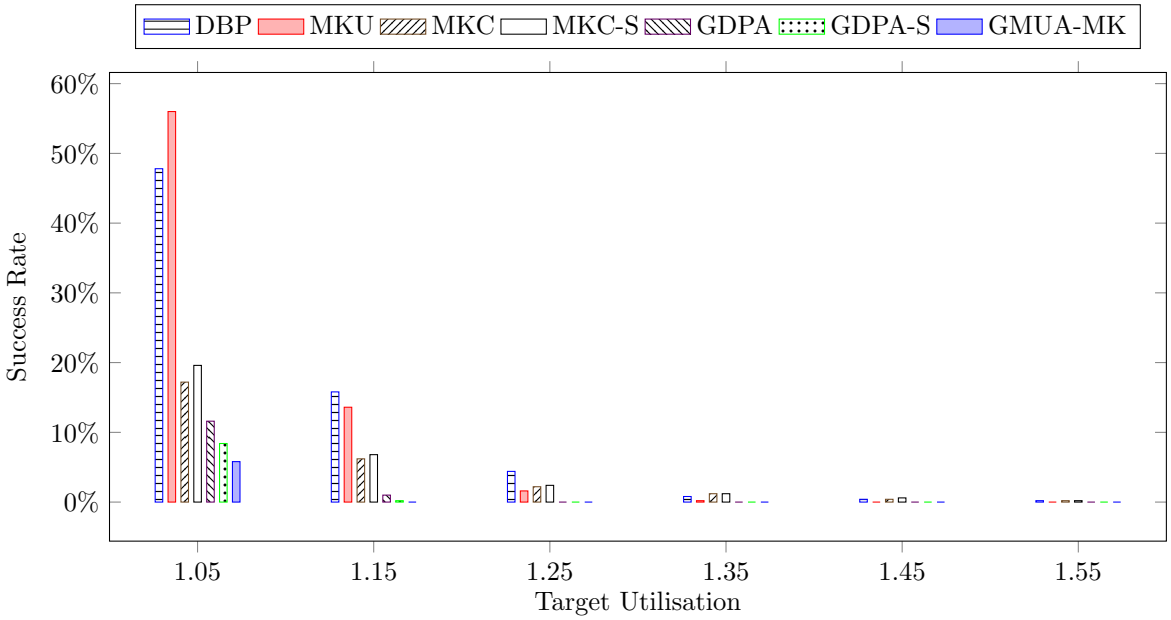



Figure 8.9.: Success rates for $r_m = 0.5$

Exemplarily, the success rates of the schedulers for $r_m = 0.5$ are shown in figure 8.9. Further diagrams for the other $r_m$ values can be found in appendix A. For most schedulers, the performance ratio between any two stays similar to that found in the above simulations. As expected, the overall performance decreases with increasing $r_m$. This can most clearly be observed by a decrease of the maximum target utilisation for which feasible CTSs exist. Also, the number of feasible CTSs at $U_T = 1.05$ decreases rapidly with increasing $r_m$.

However, the DBP and MKU schedulers exhibit a more interesting behaviour, when examined in this detail compared to the simulations in 8.5.2. For very moderate overloads ($U_T = 1.05$), MKU achieves in most simulations, where $r_m \geq 0.3$, a better average performance than DBP. It seems that in these situations the advantages of EDF, which MKU is based on, over fixed-priority scheduling can still surface despite the overload. Applying the

cross-initialisation technique (see sect. 8.5.2) to transfer successful $k$-sequences from MKU to the more runtime-efficient DBP does not yield any successes for $r_\mathrm{m} \geq 0.6$, which defines the most relevant range of $m$ parameters.

### Realistic Periods

In real applications, periods are often harmonic or have at least many common divisors. Also, they usually span several orders of magnitude. To imitate such circumstances, the period generator proposed by Goossens and Macq (J. Goossens and Macq 2001) is used, which is aimed to generate task sets with limited hyperperiod (see sect. 8.4.2). For these experiments, the matrix displayed in figure 8.10 is used as data basis. With this matrix, periods are in a range from 3 to 1,664,863,200. The maximum hyperperiod is also 1,664,863,200. All other task parameters are chosen as in the breakdown search (table 8.5).

$$\begin{pmatrix} 1 & 1 & 2 & 2 & 4 & 4 & 8 & 8 & 16 & 32 \\ 1 & 1 & 3 & 3 & 3 & 3 & 9 & 9 & 9 & 9 \\ 1 & 1 & 1 & 5 & 5 & 5 & 5 & 5 & 25 & 25 \\ 7 & 7 & 7 & 7 & 7 & 7 & 7 & 7 & 7 & 49 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 11 & 11 & 11 \end{pmatrix}$$

Figure 8.10.: Matrix used for generation of realistic task periods

Simulations are again performed for 500 ATSs. Figure 8.11 shows the ratio of ATSs that are feasible for a given target utilisation under the DBP and MKU approaches. Values for $U_\mathrm{T} > 2.05$ are omitted, as they are very small.

The larger range periods seem to have a rather detrimental effect on most schedulers. Compared to the period used in section 8.5.2, their performance is more than halved. However, some exceptions exist: For a moderate overload at $U_\mathrm{T} = 1.05$, GMUA-MK actually improves, but deteriorates fast for larger $U_\mathrm{T}$. The MKP and MKP-S schedulers actually achieve much better results. These can be attributed to the larger range of periods in the single task sets. As both approaches also use rate-monotonic priorities (C. L. Liu and Layland 1973) for their mandatory jobs, the critical instance at time $t = 0$ is greatly relieved: In the evaluation that used arbitrary (sect. 8.5.2), all jobs released at this time have similar deadlines and thus compete for processing time in the same interval. In contrast, with the extended period ranges, low-priority tasks (having long deadlines) can profit from multiple activations of tasks with shorter period within their period, as they can easily supersede the optional instances of the short-period tasks. The deterioration of DBP is founded on the fact that it often makes decisions that are suboptimal for such task sets. Depending on $(m, k)$-constraints, it happens that DPB prefers a task with higher period over one with lower period due the priority assignment being solely based on distance from dynamic failure. In such task sets, this often leads to multiple consecutive low-period jobs (with actually high RM priority) being not executed at all and thus a violation of $(m, k)$-constraints.

### Combination

Finally, the combination of restricting tasks' $m$ parameters and periods is examined. Task parameters are generated as in section 8.5.3 except for the periods, for which Goossens' and

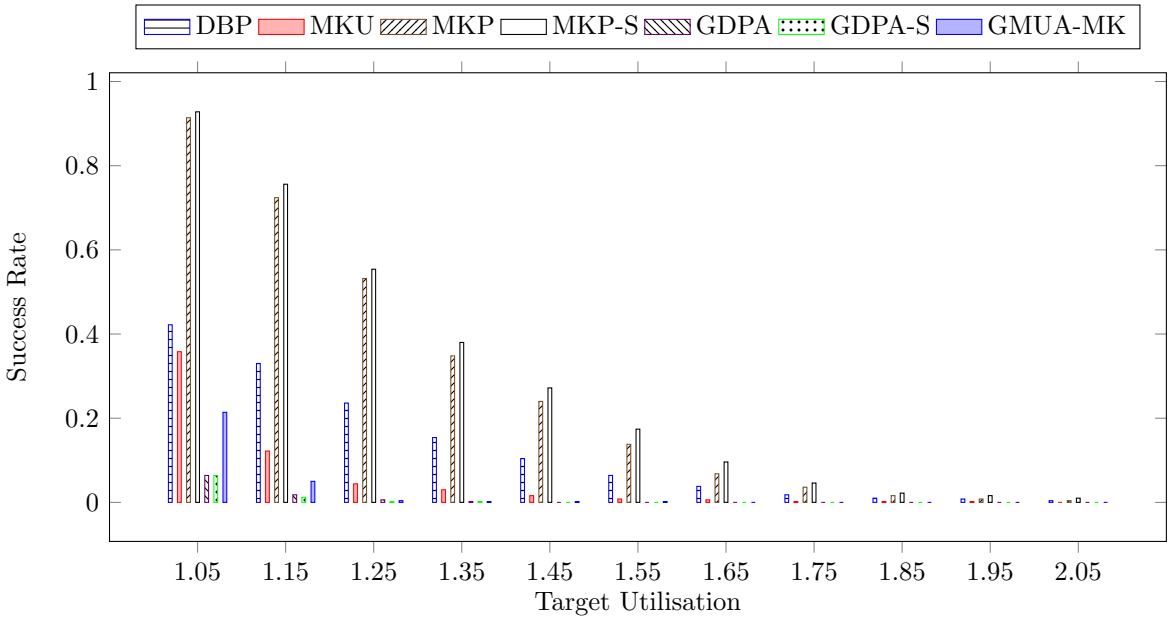



Figure 8.11.: Ratio of task sets with realistic periods that are feasible up to a certain target utilisation $U_{\mathrm{T}}$

Macq's approach (J. Goossens and Macq 2001) already used in section 8.5.3 is employed. Again, 500 ATSs are generated and simulated. Performance numbers are again based on the breakdown utilisations, ignoring the rarely occuring breakdown anomalies.

The resulting numbers can be found in appendix A. They can be interpreted as a combination of the results of the previous two experiments. The performance of all approaches is clearly dominated by the larger variance of periods. The MKP/MKP-S approaches still achieve outstanding performance due to the higher variation of task periods within the task sets. The other approaches suffer from both the regular periods and the high $r_{\mathrm{m}}$ parameter.

### 8.5.4. Discussion

The results make several points in regard to which scheduler should be used for which kind of set of $(m, k)$-firm real-time tasks. First, if task periods span several orders of magnitude, as is often the case for industrial applications, the schedulers based on fixed $(m, k)$-patterns can achieve better performance (see sect. 8.5.3). They have the additional advantage that a simple schedulability test (Jia et al. 2007) is available. If task periods can be tuned to be harmonic, the test yields exact results. Second, for task periods that are in the same order of magnitude, better results are achieved using one of the DBP or MKU schedulers (see sect. 8.5.2). Depending on how strongly the task set is constrained by $(m, k)$-parameters, either one of the two tends to yield better results. If constraints are very harsh, i.e. if the $m_i$ are very near to the $k_i$, then the performance tends to be higher under MKU, and vice versa for DBP. However, it may still happen that, e.g. a strongly constrained task set is feasible under DBP, but not under MKU. So the final choice of a scheduler must be based on a accurate examination of the task set considering all schedulers available.

## 8.6. Summary

The work presented in this chapter examines the HCUF-based scheduling of $(m, k)$-firm real-time tasks and compares it with several other schedulers. For existing schedulers for $(m, k)$-firm real-time tasks, some new properties are pointed out, namely an exact schedulability test for scheduling based on fixed $(m, k)$-patterns and the existence of breakdown anomalies in approaches like DBP. Concerning the HCUF-based heuristic MKU, formal results on the schedulability are discussed.

The experimental evaluation examines the schedulers under several points of view. Therefore, extensive simulations of randomly generated task sets are performed using the different schedulers and different generation approaches. The simulations are based on the search for breakdown utilisation (Lehoczky et al. 1989) of abstract task sets. The results show that the HCUF-based heuristic MKU can achieve a similar performance as DBP (Hamdaoui and Ramanathan 1995), which has the best performance among all schedulers regarded if task periods within a task set are roughly in the same order of magnitude and $(m, k)$-constraints are very heterogeneous. Both approaches are able to find feasible schedules for up to 80 % of the generated task sets. They also show the advantage of the optimisation that GST (J. Goossens 2008) introduces for testing the exact schedulability condition for $(m, k)$-firm real-time task sets under DBP, as GST can reduce the simulation time significantly. The results show further that no clear relation exists between a task sets' $(m, k)$-utilisation and its feasibility. Although breakdown anomalies occur during the simulations, they are very rare and thus are of rather low practical relevance. The problem of finding good initialisations of tasks' $k$-sequences is also tackled, as these can impact feasibility (J. Goossens 2008). Using results produced by a feasible schedule as initialisation for a task set under another scheduler, where it so far is infeasible, can yield only minor improvements. The examination of processing time lost due to job cancellations gives some surprising results: Under this metric, the GDPA/GDPA-S (Cho, Chung, et al. 2010) and GMUA-MK (Rhu et al. 2011) schedulers achieve best performance (less than 2 % loss), while DBP can lose up to 10 % of the processing time. The MKU approach lies somewhere between these numbers.

In further simulations, task set generation is restricted to realistic parameters. A lower bound for the $m$ parameter prohibits the generation of tasks whose jobs are scarcely executed. As can be expected having the $m_i$ paramters of tasks near their $k_i$ paramters results in a decrease of all schedulers' performances. However, it can be observed that in such a scenario with strong $(m, k)$-constraints, MKU can actually achieve better results than DBP if the overload is only moderate. By using the period generator proposed by Goossens and Macq (J. Goossens and Macq 2001), periods spanning multiple orders of magnitude inside a task set are generated. In such task sets, the performance of MKU and DBP degrades significantly due to suboptimal decisions. Concurrently, the schedulers based on fixed $(m, k)$-patterns, i.e. MKP (Ramanathan 1999) and MKP-S (Quan and Hu 2000), can achieve much higher performance (feasibility for up to $\approx 93$ % of the generated CTSs). However, it must also be noted that all results are based on averages over many task sets. Although these numbers favour certain schedulers for certain task set characteristics, they can only provide a guideline for the choice of a scheduler for a concrete task set. Actual performance still depends on the concrete task parameters, and it may happen that for the concrete use case a different scheduler can achieve better performance than the one favoured by the overall results presented in this chapter.

# Part III.

# Outlook & Conclusions

# 9 Ongoing and Future Work

The suitability of an OS for SCSs depends not only on its architecture and the general concepts it is based on, but mainly on aspects of its implementation. In section 5.8, a number of such aspects are highlighted. Some of these have been tackled in the previous chapters. This chapter gives an overview of aspects that follow from section 5.8 that are still open, and points out approaches that might be taken. In the following section 9.1, the coordination between real-time tasks is examined with the aim to achieve a more predictable timing behaviour. Problems and opportunities for applications consisting of interacting modules deployed to multiple nodes on a manycore processor are presented in section 9.2. Section 9.3 discusses the exploitation of pseudoperiodic behaviour as it is formalised in the GTM (see sect. 7.4). Further applications for HCUF-based scheduling are described in section 9.4. The need for appropriate benchmark applications is explained in section 9.5. While the basic MOSSCA implementation is based on a research prototype of a manycore processor, it should also be investigated how its concepts can be applied in existing industrial manycore processors. A brief overview of near-future goals for this topic is given in section 9.6.

## 9.1. Coordination

### 9.1.1. Background

Each design and implementation of a real-time application is based on a specific *execution model (EM)*. The EM specifies semantics about program execution, concerning e.g. the timing correctness, or how and when communication takes place. In the following, the most important models are introduced only briefly, refer to (Kirsch 2002; Kirsch and Sengupta 2007) for a more extensive description. The basic definitions of these models are based on tasks implementing the input-process-output (IPO) model. Most widely used is the *bounded execution time (BET)* model. In the BET model, a (periodic) task is considered to execute correctly, if input, processing and output of one instance take place before the task's next activation, i.e. within the task's bounded execution time (Kirsch and Sengupta 2007). While enabling the use of multitasking and being easy to implement, the BET model has also some drawbacks. First, this approach has predictability problems if critical sections are used, as task scheduling is prone to priority inversion and deadlocks. To get these under control, additional mechanisms like *priority inheritance* or the *priority ceiling protocol* (Sha et al. 1990)

can be employed. Additionally, the BET model is lacking compositionality with respect to I/O: Changing a task set by adding or removing tasks can lead to changes in the schedule for the other tasks. The result are changes of I/O times and possibly also of I/O values. Thus, changing the task set can lead to a different behaviour of tasks that actually are not affected by the changes.

One step to overcome these drawbacks was the introduction of the *synchronous reactive model (SRM)* (Nicolas Halbwachs 1992; Potop-Butucaru et al. 2005), also termed *zero execution time (ZET)* (Kirsch and Sengupta 2007). Most central aspects of the SRM are that task inputs are always read at the same time at the release of a task, and that actual processing consumes no time. Thus, I/O compositionality is guaranteed. In synchronous-reactive languages like Lustre (Nicholas Halbwachs et al. 1991) or Esterel (Berry 2000), applications are modeled in a parallel manner. The languages' semantics often allow to formally proof the correctness of programs. The programs are compiled into sequential code that is free from any concurrency problems. This is also a slight drawback, as compositionality can only be utilised during design time. Another drawback is the ZET assumption which can pose high performance requirements on OS and underlying hardware to be realised at least approximately. The implementation of SRM applications in distributed systems where communication latencies cannot be neglected is even more complex (Kirsch and Sengupta 2007), although some solutions exist (Benveniste et al. 2002; Caspi et al. 2003).

With the timing language Giotto (Henzinger et al. 2003), the concept of *logical execution time (LET)* was introduced. LET semantics assume that task inputs are read at the beginning of a task's LET and outputs are written at the end of the LET. Actual task execution can take place anywhere within the LET. As long as a task set schedulable, full compositionality is achieved by the LET approach. Further languages that implement LET are xGiotto (Ghosal et al. 2004), HTL (Ghosal et al. 2006), and TDL (Templ 2004; Pree and Templ 2006). Compared to the SRM, the LET model is based on more realistic design assumptions. Although is does not use fixed-point semantics as the SRM, it still yields a well-defined application behaviour, e.g. through predictable communication times. A drawback of the LET concept is that it allows only periodic tasks, except for xGiotto (Ghosal et al. 2004) which can also handle aperiodic tasks. The delay of one period length between input and corresponding output can also deteriorate the performance of control systems (Åström and Wittenmark 1990; Shin and Cui 1995; Wittenmark et al. 1995). Appropriate techniques allow to consider the execution delay in controller design (Mita 1985; Fontanelli et al. 2013), thus solving this problem. Furthermore, applications developed according to the LET concept cannot exploit increasing processor performance, as I/O times are fixed at task design. Nevertheless, additional performance can be used to execute further applications on the same processor.

### 9.1.2. Support for LET in MOSSCA

MOSSCA as defined in chapter 5 only supports applications that are executed according to the BET model. By restricting all communication between tasks to explicit messages, problems that may arise from scheduling anomalies are circumvented. The GTM (ch. 7) provides basic means that may be used to improve worst-case response times for server usage for a concrete MOSSCA system consisting of platform and application. Nevertheless, the next logical step is to improve compositionality of a MOSSCA system (see also sect. 5.8.5).

The MOSSCA abstractions (sect. 5.2) shall be extended to support applications implemented according to the LET concept. The main focus lies on the extension of the *Channel*

abstraction, to ensure timely availability of messages. Node-local communication can be reduced to single-core execution of LET-based applications, which can be solved using existing approaches as implemented in OASIS (Aussagués and David 1998; David et al. 1998) or Giotto (Henzinger et al. 2003). Communication between tasks running on different nodes requires that also NoC latencies are taken into account when calculating task schedules resp. scheduling parameters.

A prototype implementation will be performed using the MOSSCA port for the T-CREST platform (see sect. 5.5.3). First application examples will be implemented in a cyclic executive manner that is also generated by Giotto: A static schedule is generated for tasks executed on the same node, thereby executing each task non-preemptively. Later implementations shall also be able to utilise dynamic schedulers that enable a better utilisation of the hardware. The prototype will allow to evaluate the overhead that is introduced through the LET extension. Furthermore, different implementation approaches (e.g. global shared memory vs. scratchpad memories) for different communication patterns (e.g. 1:1 or broadcast messages) can be compared.

## 9.2. Scheduling of Computation Chains

Applications in a MOSSCA system typically implement reactive computations. As parallelisation of computations is inevitable, each the computation of a reaction will be decomposed into sub-computations that are executed in a parallel or pipelined pattern. While data parallelism helps to increase the actual execution speed, pipeline parallelism increased the computational throughput. Figure 9.1 exemplarily shows a pipelined reaction. The upper part considers the model as introduced in GTM (sect. 7.2). At the start of a reaction, input data is read by a sensor component. The data is processed in two subsequent computation components. Finally, outputs are set through an actuator component. The lower part of figure 9.1 considers the mapping of the components to a MOSSCA system. Sensor and actuator components are managed by I/O servers (these may be identical), while computations are performed on application nodes. To maximise throughput, the components should be distributed to different nodes as far as possible.

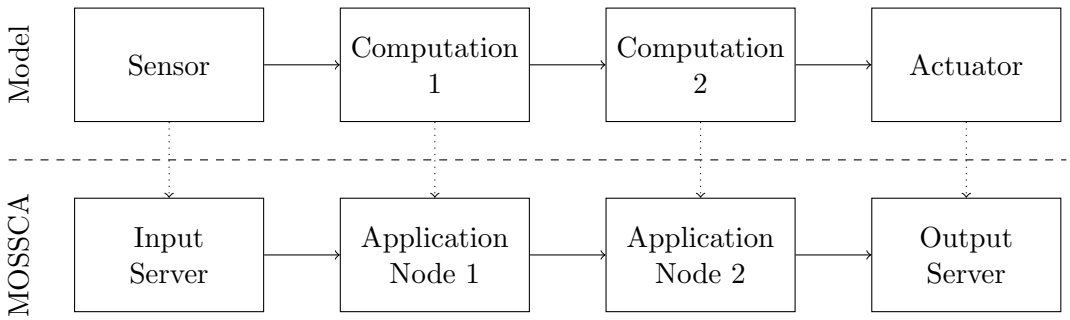



Figure 9.1.: Parallelisation model and mapping (⋯›) to MOSSCA system

Despite the demand for distribution, it is to be expected that nodes will have to execute multiple tasks using node-local schedulers to increase processor utilisation and system flexibility. Single nodes, especially I/O servers thus might be shared by multiple computation chains. This opens up to aspects which are discussed in the next sections. The (sub-)tasks

that are executed by the node-local schedulers need to be assigned deadlines to ensure timely execution of the whole reactions (sect. 9.2.1). Knowledge about the arrival times of subtasks might be exploited to reduce output jitter (sect. 9.2.2).

### 9.2.1. Deadline Assignment

A typical approach during application development is to map full reactions (from input to output) to periodic tasks (C. L. Liu and Layland 1973) which have deadlines equal to their periods. It is to be expected that this mapping will also be kept when a computation is split into multiple sub-computations that are executed on different application nodes. Each sub-computation needs to be assigned appropriate scheduling parameters such that the whole reaction is executed in a timely manner. More specifically, a *deadline assignment* for the sub-computations has to be performed. This problem is well-known in the area of distributed real-time systems (see e.g. Jonsson and Shin 1997; Hong et al. 2011). The applicability of existing approaches should be investigated.

### 9.2.2. Server Reservations for Response Time Improvements

MOSSCA servers perform their services in the form of non-interruptable transactions. From the viewpoint of scheduling theory, they execute schedules consisting of non-preemptive jobs. The general problem of scheduling jobs non-preemptively is NP-complete (Garey and Johnson 1979; George et al. 2000). Also, response time analysis and determination of output jitter are much harder than for preemptive systems. A concrete problem of non-preemptive scheduling is that inserting idle times in a schedule can actually improve schedulability (Howell and Venkatrao 1995). If all jobs are known in advance, a clairvoyant scheduler can insert idle times to derive a feasible scheduler (Ekelin 2006).

The distributed manner, in which reactions are performed in a MOSSCA system allows for an optimisation: Once the input event for a reaction happened, the process until output is fixed and known. As all sub-tasks have undergone a WCET analysis, one can estimate their actual activation times immediately after the input event. This information can be used in the I/O server that performs the output to reduce the output jitter of at least some computation chains. Therefore, the input server or a subsequent component of the computation chain notifies the output server about the arrival of a reaction. From the WCET analysis results, an arrival time window of the request at the output server can be estimated. The output server places a *server reservation* for this time window in its queue of incoming requests. Requests arriving in the meantime are only served if they cannot delay the request specified by the server reservation. Insofar, the server reservation is a hint to the server scheduler on when to insert idle times in the scheduler. The most pressing question is whether a schedulability analysis for this approach is feasible. Furthermore, an experimental evaluation should examine how deviations from (sub-)task WCETs influence the server performance. Another interesting parameter is the degradation of QoS experienced by reactions that do not use server reservations but share the output server.

## 9.3. Exploitation of Pseudoperiodic Behaviour

Recall example 7.14 from section 7.4 that explains the modeling of pseudoperiodic events in the GTM. In this example, the signal trace generated by a crank shaft sensor of a combustion

engine is examined. Events in this trace occur pseudoperiodically with frequency changes: As long as the engine speed is constant, the events occur with strict periods. When the speed changes, the periods also change. As the events trigger computations in the EMS, their frequency directly influences the current utilisation of the platform the EMS is executed on. A low speed thus is equivalent to low utilisation, meaning that there might be sufficient processing time for additional tasks. Changes of the engine's speed are subject to physical constraints, which in GTM are expressed through bounds $\pi_{\min}, \pi_{\max}$ on the elements $\pi_i$ of a frequency change sequence $\Pi(E)$. At any moment in time, these a bound for the amount of free processing time in the short-term future can be predicted based on the current engine speed and the bounds $\pi_{\min}, \pi_{\max}$.

To build more flexible systems, it might be profitable to use processing time that is available during intervals of low utilisation to execute other tasks, e.g. in a distributed real-time system where tasks can be migrated between nodes. An acceptance test on the single nodes can use the prediction of free processing time for its decision. The deployment of a task that is executed in this manner makes only sense, if certain guarantees for its execution can be given. Further investigation should examine these in more detail. An obvious approach is the application of $(m, k)$-firm real-time constraints. Using fixed $(m, k)$-patterns (Ramanathan 1999), execution of mandatory jobs must be guaranteed, while the optional instances would be candidates for the acceptance test. Also interesting might be a combination with scheduling servers like the total bandwidth server (TBS) Spuri and G. C. Buttazzo 1994; Spuri and G. C. Buttazzo 1996 or the constant bandwidth server (CBS) Abeni and Giorgio Buttazzo 1998; Giorgio Buttazzo and Bini 2006.

## 9.4. HCUF-Based Scheduling

Scheduling based on HCUFs can be expanded beyond $(m, k)$-firm real-time tasks. The experiments described in chapter 8 demonstrate the feasibility of using HCUFs for scheduling. However, HCUFs can not only be used to map $(m, k)$-constrains, they may convey any kind of information. Especially, it will be interesting to use HCUFs and possibly further means to implement an interaction between a scheduler and applications. Thus, application could report changing requirements to the scheduler, which thus could adjust schedules according to the current situations. Vice versa, the scheduler could also provide information about its decisions to an applications, which thus would be able, e.g. to adjust its behaviour. Such a scheduler could be used, e.g. for control systems. Hitherto, controller are mostly designed using tight timing constraints, which must be ensured by the scheduler. If actual execution deviates too far from these constraints, especially if the controlled system is undergoing change, the quality of control can degrade. If the controlled system is in a steady state, timing constraints might be seen more relaxed. Enabling communication between scheduler and control application would allow to adjust the execution rate according to the current situation of the control system, thus e.g. saving performance during steady states.

## 9.5. Benchmarking

Numerous scientific projects are engaged in making manycore processor usable in real-time domains. Predictable processor architectures, operating systems, and also tools and techniques for parallelisation or WCET analysis are objects of research. To evaluate the developed tech-

niques, researchers depend on the existence of appropriate test and benchmark programs. For example, the *Mälardalen WCET Benchmarks* (Gustafsson et al. 2010) are widely used in the domain of WCET analysis. Execution of parallelised programs can be examined with the *PARSEC Benchmark Suite* (Bienia 2011).

Such and similar suites have the drawback that the contained programs do not map the complexity of today's RTES. Thus, they cannot be used to extensively evaluate a system architecture. Also, they do not exhibit reactive behaviour, which is an important property of RTES. So far, only few works exist that aim to close this gap. PapaBench (Nemer et al. 2006) is derived from the Paparazzi project[1]. The Paparazzi project builds an unmanned aerial vehicle (UAV). PapaBench thus resembles a real-world real-time application consisting of concurrent tasks and interrupt handlers.

Another step towards system benchmarks was taken during this work with the development of EMSBench (Kluge and Ungerer 2015). EMSBench provides a benchmark program based on the open source EMS FreeEMS[2], and a testbed that generates relevant input signals and thus allows to execute the benchmark program. The benchmark code derived from FreeEMS consists mainly of multiple ISRs that interact among each other to control fuel injection and ignition in a combustion engine. Even though FreeEMS may not have the complexity of actual industrial code for engine management, it still exhibits a similar behaviour and is more complex than simple linear program execution. Currently, EMSBench is designed to run on a single-core processor. To facilitate real-time research in multi-/manycore processors, EMSBench shall be extended to run on a multicore processor. Also, a port to the MOSSCA platform is envisaged.

## 9.6. Investigation of Existing Manycore Platforms

The MOSSCA reference implementation is based on a simulated manycore processor. So far, MOSSCA has only been ported to the T-CREST manycore platform running on an FPGA (see sect. 5.5.3). It will definitely be rewarding to investigate how MOSSCA or its core concepts can be applied existing industrial manycore processors, e.g. to investigate performance tradeoffs.

Most promising candidates for such ports are the Epiphany platform (Adapteva 2013c) and the Kalray MPPA-256 manycore (Dinechin et al. 2013). Concerning the Epiphany, initial experiments are being performed to investigate the timing predictability of the processor's NoC (Wermund 2015). The MPPA-256 is challenging, as it combines both shared memory and messages paradigms in a single processor.

[1] `http://www.recherche.enac.fr/paparazzi`
[2] `http://freeems.org/`

# 10 Conclusions

## 10.1. Summary

This work aims to leverage the use of manycore processors in future SCS. The focus lies on the operating systems, which plays a key role for the usability of hardware platforms and for development of future applications. The introduction in chapter 1 illustrates the trend towards integrating more and more cores, and motivates the aims of this work.

In chapter 2, the properties and requirements of SCSs are reviewed. Although the occurrence of mishaps cannot be excluded completely, engineers strive to achieve mishap probabilities of down to $10^{-9}$. In general, these are accomplished by improving reliability and quality of components, or by the integration of internal or external safety devices. In the context of computers and especially software in SCSs, improved reliability and quality are mainly that a correct functional and timing behaviour is ensured. Adding fault-tolerance properties can additionally improve reliability. In SCSs, functional and timing correctness are ensured through mechanisms that can be subsumed under the term *partitioning in space and time*. Partitioning mechanisms ensure that faults emerging in one application cannot influence or propagate to other applications running on the same computer. They are often integrated in hypervisors, thus allowing to use also different, application-specific operating systems. Chapter 2 closes with the definition of SCS requirements on OSs that are relevant for this work: An OS must (1) provide and support predictability of itself as well as of applications, provide appropriate mechanisms for (2) partitioning and (3) communication inside and between partitions, and (4) permit dynamic reconfiguration for more flexible future SCSs.

Manycore processors, especially for SCSs are examined closer in chapter 3. After a brief overview of the developments in processor architecture during the last 10 years, the architectural characteristics of manycore processors are carved out. Typical properties of manycore processors are (1) the availability of fast on-chip communication, (2) that core-local memories (with low access latencies) are rather small, and (3) that the use of large memories usually requires comparably slow off-chip communication. This chapter also discusses the special challenges for SCS software developers that are posed by multi- and manycore processors, e.g. the sharing of resources, or the existence of single points of failure, like clock distribution or power supply. The chapter closes with the definition of additional requirements for OSs that arise from the special properties of manycore processors: (1) OS code size should heed

the size of core-local memories, i.e. it should be as small as possible, and (2) sharing of large data structures between multiple cores should be avoided to attenuate interferences.

Chapter 4 reviews related work on operating systems for manycore processors. A number of existing approaches are examined. Although none of these is explicitly targeted at SCSs, all of these approaches are able to fulfil parts of the requirements of the previous chapters. Adjusting or removing mechanisms that contradict the requirements might allow to apply some of the OSs in SCSs. Also, a number of work on virtualisation in SCSs is inspected. The methods developed aim mainly at the provision of time and space partitioning. Additionally, some works also tackle the problem of memory and performance constrictions in embedded systems.

The second part of this work starts with the presentation of a concrete operating system for manycore processors in SCSs, the *manycore operating system for safety-critical applications* (MOSSCA, ch. 5). MOSSCA serves as basis for concepts examined in later chapters of this work. It is based on the concepts implemented in the *factored operating system* (Wentzlaff and Agarwal 2009, sect. 4.1.3), namely the extensive distribution of OS functionalities over the manycore processor. As underlying hardware, MOSSCA assumes a homogeneous manycore architecture consisting of simple cores with local memories. Communication between cores is performed through explicit messages that are sent over a real-time NoC. A central concept of MOSSCA is to distribute application threads as far as possible over the processor. Simply put, this could mean that each thread is assigned an own core. For application developers, MOSSCA defines several abstractions: *Nodes* represent the primary execution resource of application threads. Unidirectional *communication channels* allow the exchange of data between application threads inside a partition. Their implementation ensures that bandwidth restriction are met even if an application thread is misbehaving ("babbling idiot"). *Servers* in MOSSCA provide services that must be centralised (e.g. I/O access), or that are used by multiple threads. Usage policies ensure a predictable timing behaviour of server usage through applications. These abstractions are implemented in the architecture of MOSSCA. On each node of the manycore processor, an identical kernel is deployed that manages local resources. It also enforces the usage policies of communication channels and servers used by the local application. On top of the kernel, either applications (or application threads), or servers are implemented. For application-specific servers, i.e. I/O or library servers, a server framework is provided by MOSSCA that enables the timing analysis of server requests, e.g. by providing prioritised handling of requests. The OS Server is a special kind of server: It manages all on-chip resources and is also, e.g. responsible for (re-)configuration of the whole system. It is not directly used by the application, but certain calls to the local kernel involve interaction with the OS Server. A reference implementation of MOSSCA performed on the MacSim manycore simulator (Metzlaff et al. 2011) is presented. Additionally, some use-case implementations are discussed. Concerning an automotive environment, core services of AUTOSAR OS using the MOSSCA concepts are examined. In the parMERASA project (Ungerer et al. 2013), MOSSCA provides the initial baseline for the system software. Also, a port of MOSSCA to the time-predictable manycore platform from the T-CREST project (T-CREST 2013) is available. The challenge here is the mapping to an architecture that provides both local and global memories. Due to restricted size of local memories, the global memory is used for most parts of MOSSCA. Section 5.6 shows, how MOSSCA fulfils the requirements defined in the first part (chapters 2 and 3) of this work. It is followed by a discussion of virtualisation in the context of MOSSCA. This discussion shows how virtualisation of RTEs can be accomplished using the server concept in MOSSCA. The analysis of a MOSSCA system concerns several

aspects that are discussed in section 5.8. Among these are boostrapping, the scheduling of server requests, local multithreading, and the coordination between threads. Some of these aspects are examined in later chapters of this work.

In chapter 6, the boot process of a manycore computer is analysed in terms of its worst-case duration. The organisation of OS and application code and data plays an important role in this analysis. Three approaches are examined: In two state-of-the-art approaches, either (1) each node receives a single image containing kernel and application, or (2) kernel and application are split in separate images, where the kernel is the same for any node. As an optimisation, the use of a self-distributing kernel is proposed. Evaluation of these approaches is based on the analysis of sequential code with OTAWA (Ballabriga et al. 2010) and the simulation of the worst case of the whole boot process using the `mwsim` tool. `mwsim` executes scripts that model the boot process. Simulations are performed for several processor sizes. The results show a clear advantage for the self-distributing kernel, as this approach largely reduces the load on the NoC interface of the boot core that coordinates the whole boot process. Another important finding is that splitting images can result in a higher WCD than the full image approach in large processors. Finally, potentials for further works in this direction are discussed, e.g. by regarding the use of DMA units or a best-effort NoC.

One of the aspects influencing the analysis in a MOSSCA system is the coordination between tasks. Although, when deployed to separate cores, tasks may execute without interferences, there is still a need for coordination, e.g. when exchanging data, or when concurrently using a server. The generic timing model that is presented in chapter 7 aims to provide the means for modeling and capturing timing behaviour and timing influences between tasks in CPSs. The structure of the CPS is modeled based on components that can be composed hierarchically. Exchange of data between components is modelled through communication channels. The behaviour of a CPS is characterised by events that propagate over channels, and by reactions that components perform in response to incoming events. Recurring events and reactions are modelled as sequences. Utility functions are used to rate single reactions (time-utility functions) or reaction sequences (history-cognisant utility functions) in terms of the benefit the CPS experiences through their execution. Special heed is paid to the modeling of pseudoperiodic behaviour, as it occurs, e.g. in a combustion engine.

While the TUFs and HCUFs in GTM are originally intended for visualisation of timing properties and requirements of components, they provide also interesting means for scheduling of real-time tasks. In chapter 8, the HCUFs concept is applied for scheduling $(m, k)$-firm real-time tasks in overloaded real-time systems. Such tasks require that only at least $m$ out of any $k$ consecutive jobs are executed successfully. The MKU scheduler uses a special HCUF that is designed to map $(m, k)$-constraints. The HCUF values are used by an EDF-based scheduler to decide about cancellation of jobs in overload situations. Using simulations, MKU is compared with other schedulers for $(m, k)$-firm real-time tasks. A number of task set characteristics are identified that can act as a guideline when choosing a scheduler for a concrete set of $(m, k)$-firm real-time tasks.

Concerning the analysis aspects introduced in section 5.8, but also some that arise in later chapters, there is still work ongoing. These and additional points are discussed in chapter 9. The coordination between real-time tasks and appropriate OS support need closer examination. In the MOSSCA context, the application of LET will be examined using the port to the T-CREST platform. Additionally, computational chains in a MOSSCA system can be examined under two aspects. First, when core-local multitasking is used, it is necessary to assign single computations in such a chain deadlines such that end-to-end deadlines of the chain can

be met. Second, the concept of server reservations, where time slots on servers are reserved in advance, should be examined. Concerning GTM, there are further ways to exploit the models. One important point would be the use of knowledge about pseudoperiodic behaviour, e.g. for an acceptance test of aperiodic tasks. Another point is the continuation of using HCUFs provided through GTM for scheduling. Next, the problem of system benchmarks is tackled. While there are many benchmark suites for real-time systems that consist of multiple small programs, only few benchmarks representing the actual complexity of today's systems exist. In the context of this work, the system benchmark EMSBench was developed. EMSBench is based on an open source engine management software. Evaluations with EMSBench and its further development towards parallel systems are still ongoing. Finally, existing commercial manycore platforms should be examined more closely.

## 10.2. Conclusions

This work contributes to several aspects in the domains of SCS or more generally embedded real-time systems. The MOSSCA concepts and architecture provide a sound baseline for developing applications in SCS. Their feasibility is demonstrated through multiple implementations. Important OS mechanisms are developed and examined. While having received only minor attention in the past, the boot process of a manycore processor is investigated especially from a timing point of view. Problems of resource sharing are tackled in several places. The GTM provides means to model timing behaviour and properties of applications in CPS in a hierarchical manner. Information captured with resp. provided by GTM can also be used to improve scheduling. This is demonstrated using the example of $(m, k)$-firm real-time tasks. The MKU scheduler developed in this work resolves overload situations by cancelling jobs, thereby heeding information that is conveyed through HCUFs. Finally, this work identifies several points for future investigations. Some of these are closely related to manycore processors the MOSSCA platform, like the coordination of tasks or scheduling of server requests and computation chains. Others, like the exploitation of pseudoperiodic behaviour or HCUF-based feedback between scheduler and applications, can be seen in a broader context.

# Bibliography


Abbaspour, Sahar, Florian Brandner, and Martin Schoeberl (2013). "A time-predictable stack cache". In: *16th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing, ISORC 2013, Paderborn, Germany, June 19-21, 2013*, pp. 1–8. DOI: 10.1109/ISORC.2013.6913225 (cit. on p. 47).

Abeni, Luca and Giorgio Buttazzo (1998). "Integrating multimedia applications in hard real-time systems". In: *The 19th IEEE Real-Time Systems Symposium, 1998. Proceedings.* Pp. 4–13. DOI: 10.1109/REAL.1998.739726 (cit. on p. 123).

Adapteva (2013a). *E16G301 Epiphany™16-Core Microprocessor Datasheet*. 14.2.21. Adapteva, Inc. Lexington, MA, USA (cit. on p. 4).

— (2013b). *E64G401 Epiphany™64-Core Microprocessor Datasheet*. 14.03.11. Adapteva, Inc. Lexington, MA, USA (cit. on p. 4).

— (2013c). *Epiphany Architecture Reference*. 14.2.21. Adapteva, Inc. Lexington, MA, USA (cit. on pp. 11, 13, 33, 37, 55, 124).

— (2013d). *Epiphany SDK Reference*. 5.13.09.10. Adapteva, Inc. Lexington, MA, USA (cit. on p. 37).

Albers, Karsten, Frank Bodmann, and Frank Slomka (2006). "Hierarchical event streams and event dependency graphs: a new computational model for embedded real-time systems". In: *18th Euromicro Conference on Real-Time Systems, 2006.* Pp. 97–106. DOI: 10.1109/ECRTS.2006.12 (cit. on p. 74).

Aldarmi, Saud A. and Alan Burns (1999). "Dynamic value-density for scheduling real-time systems". In: *Proceedings of the 11th Euromicro Conference on Real-Time Systems, 1999.* Pp. 270–277. DOI: 10.1109/EMRTS.1999.777474 (cit. on p. 89).

Anderson, Thomas E., Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy (1992). "Scheduler activations: effective kernel support for the user-level management of parallelism". In: *ACM Trans. Comput. Syst.* 10.1, pp. 53–79. ISSN: 0734-2071. DOI: 10.1145/146941.146944 (cit. on pp. 18, 39).

Asanovic, Krste, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick (2006). *The Landscape of Parallel Computing Research: A View from Berkeley*. Tech. rep. UCB/EECS-2006-183. EECS Department, University of California, Berkeley (cit. on pp. 4, 11).

Asanovic, Krste, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick (2009). "A view of the parallel computing landscape". In: *Commun. ACM* 52.10, pp. 56–67. ISSN: 0001-0782. DOI: 10.1145/1562764.1562783 (cit. on p. 11).

Åström, Karl J. and Björn Wittenmark (1990). *Computer-controlled Systems: Theory and Design (2Nd Ed.)* Upper Saddle River, NJ, USA: Prentice-Hall, Inc. ISBN: 0131686003 (cit. on p. 120).

Audsley, Neil, Alan Burns, Mike Richardson, Ken Tindell, and Andy J. Wellings (1993). "Applying new scheduling theory to static priority pre-emptive scheduling". In: *Software Engineering Journal* 8.5, pp. 284–292. ISSN: 0268-6961 (cit. on p. 92).

Aussagués, Christophe and Vincent David (1998). "A method and a technique to model and ensure timeliness in safety critical real-time systems". In: *Engineering of Complex Computer Systems, 1998. ICECCS '98. Proceedings. Fourth IEEE International Conference on*, pp. 2–12. DOI: 10.1109/ICECCS.1998.706651 (cit. on p. 121).

AUTOSAR (2011). *Specification of Operating System*. Version 5.0.0, Release 4.0 Rev 3. AUTOSAR Development Cooperation (cit. on pp. 42, 45).

Baldin, Daniel and Timo Kerstan (2009). "Proteus, a Hybrid Virtualization Platform for Embedded Systems". In: *Analysis, Architectures and Modelling of Embedded Systems, Third IFIP TC 10 International Embedded Systems Symposium, IESS 2009, Langenargen, Germany, September 14-16, 2009. Proceedings*. Ed. by Achim Rettberg, Mauro Cesar Zanella, Michael Amann, Michael Keckeisen, and Franz J. Rammig. Vol. 310. IFIP Advances in Information and Communication Technology. Springer, pp. 185–194. ISBN: 9783642042836. DOI: 10.1007/978-3-642-04284-3_17 (cit. on p. 27).

Ballabriga, Clément, Hugues Cassé, Christine Rochange, and Pascal Sainrat (2010). "OTAWA: an Open Toolbox for Adaptive WCET Analysis". In: *Software Technologies for Embedded and Ubiquitous Systems - 8th IFIP WG 10.2 International Workshop, SEUS 2010, Waidhofen/Ybbs, Austria, October 13-15, 2010. Proceedings*. Springer, pp. 35–46. DOI: 10.1007/978-3-642-16256-5_6 (cit. on pp. 57, 127).

Barham, Paul, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield (2003). "Xen and the Art of Virtualization". In: *SIGOPS Operating Systems Review* 37.5, pp. 164–177. ISSN: 0163-5980. DOI: 10.1145/1165389.945462 (cit. on p. 28).

Baruah, Sanjoy K., Gilad Koren, Bhubaneswar Mishra, Arvind Raghunathan, Louis Rosier, and Dennis Shasha (1991). "On-line scheduling in the presence of overload". In: *32nd Annual Symposium on Foundations of Computer Science, 1991. Proceedings*. Pp. 100–110. DOI: 10.1109/SFCS.1991.185354 (cit. on p. 89).

Baumann, Andrew, Paul Barham, Pierre-Évariste Dagand, Timothy L. Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania (2009). "The Multikernel: A new OS architecture for scalable multicore systems". In: *22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*. Big Sky, Montana, USA: ACM, pp. 29–44. DOI: 10.1145/1629575.1629579 (cit. on pp. 17, 18, 46).

Baumann, Andrew, Simon Peter, Adrian Schüpbach, Akhilesh Singhania, Timothy Roscoe, Paul Barham, and Rebecca Isaacs (2009). "Your computer is already a distributed system. Why isn't your OS?" In: *HotOS'09: 12th Workshop on Hot Topics in Operating Systems, May 18-20, 2009, Monte Verità, Switzerland* (cit. on pp. 17, 31).

Belay, Adam (2011). "Message Passing in a Factored OS". MA thesis. Massachusetts Institute of Technology (cit. on p. 20).

Bell, Ron (2006). "Introduction to IEC 61508". In: *Proceedings of the 10th Australian Workshop on Safety Critical Systems and Software - Volume 55*. SCS '05. Sydney, Australia: Australian Computer Society, Inc., pp. 3–12. ISBN: 1920682376 (cit. on p. 7).

Benveniste, Albert, Paul Caspi, Paul Le Guernic, Hervé Marchand, Jean-Pierre Talpin, and Stavros Tripakis (2002). "A Protocol for Loosely Time-Triggered Architectures". In: *Embedded Software, Second International Conference, EMSOFT 2002, Grenoble, France, October 7-9, 2002, Proceedings*, pp. 252–265. DOI: 10.1007/3-540-45828-X_19 (cit. on p. 120).

Bernat, Guillem, Alan Burns, and Alberto Liamosi (2001). "Weakly hard real-time systems". In: *IEEE Transactions on Computers* 50.4, pp. 308–321. ISSN: 0018-9340. DOI: 10.1109/12.919277 (cit. on pp. 5, 78, 82, 87).

Berry, Gérard (2000). "The foundations of Esterel". In: *Proof, Language, and Interaction, Essays in Honour of Robin Milner*. Ed. by Gordon D. Plotkin, Colin Stirling, and Mads Tofte. The MIT Press, pp. 425–454. ISBN: 9780262161886 (cit. on p. 120).

Bershad, Brian N., Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy (1991). "User-level interprocess communication for shared memory multiprocessors". In: *ACM Transactions on Computer Systems* 9.2, pp. 175–198. ISSN: 0734-2071. DOI: 10.1145/103720.114701 (cit. on p. 18).

Bhattacharyya, Shankar P., Hervé Chapellat, and Lee H. Keel (1995). *Robust Control: The Parametric Approach*. Prentice-Hall information and system sciences series. Prentice Hall PTR. ISBN: 9780137815760 (cit. on p. 87).

Bienia, Christian (2011). "Benchmarking Modern Multiprocessors". PhD thesis. Princeton University (cit. on p. 124).

Bjerregaard, T. and J. Sparso (2005). "A router architecture for connection-oriented service guarantees in the MANGO clockless network-on-chip". In: *Proceedings of Design, Automation and Test in Europe (DATE), 2005*, 1226–1231 Vol. 2. DOI: 10.1109/DATE.2005.36 (cit. on p. 32).

Black, B., M. Annavaram, N. Brekelbaum, J. DeVale, Lei Jiang, G.H. Loh, D. McCauley, P. Morrow, D.W. Nelson, D. Pantuso, P. Reed, J. Rupley, Sadasivan Shankar, J. Shen, and C. Webb (2006). "Die Stacking (3D) Microarchitecture". In: *39th Annual IEEE/ACM International Symposium on Microarchitecture, 2006. MICRO-39.* Pp. 469–479. DOI: 10.1109/MICRO.2006.18 (cit. on pp. 13, 32).

Blom, Hans, Rolf Johansson, and Henrik Lönn (2009). "Annotation with Timing Constraints in the Context of EAST-ADL2 and AUTOSAR – the Timing Augmented Description Language". In: *Workshop on the Definition, evaluation, and exploitation of modelling and computing standards for Real-Time Embedded Systems 2009 (STANDRTS'09)*. Dublin, Ireland (cit. on p. 74).

Boniol, Frédéric, Hugues Cassé, Eric Noulard, and Claire Pagetti (2012). "Deterministic Execution Model on COTS Hardware". In: *Architecture of Computing Systems (ARCS 2012)*. Ed. by Andreas Herkersdorf, Kay Römer, and Uwe Brinkschulte. Vol. 7179. Lecture Notes in Computer Science. 10.1007/978-3-642-28293-5_9. Springer Berlin / Heidelberg, pp. 98–110. ISBN: 9783642282928. DOI: 10.1007/978-3-642-28293-5_9 (cit. on pp. 4, 12).

Boyd-Wickizer, Silas, Haibo Chen, Rong Chen, Yandong Mao, Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yuehua Dai, Yang Zhang, and Zheng Zhang (2008). "Corey: An Operating System for Many Cores". In: *Proceedings of the*

*8th USENIX conference on Operating systems design and implementation*. OSDI'08. San Diego, California: USENIX Association, pp. 43–57 (cit. on p. 15).

Bradatsch, Christian, Florian Kluge, and Theo Ungerer (2013). "A Cross-Domain System Architecture for Embedded Hard Real-Time Many-Core Systems". In: *High Performance Computing and Communications 2013 IEEE International Conference on Embedded and Ubiquitous Computing (HPCC/EUC), 2013 IEEE 10th International Conference on*, pp. 2034–2041. DOI: 10.1109/HPCC.and.EUC.2013.293 (cit. on p. 47).

— (2014). "Comparison of Service Call Implementations in an AUTOSAR Multi-core OS". In: *9th IEEE International Symposium on Industrial Embedded Systems (SIES 2014)*. Pisa, Italy (cit. on p. 47).

— (2016). "Data Age Diminution in the Logical Execution Time Model". In: *29th GI/ITG International Conference on Architecture of Computing Systems (ARCS 2016), Nuremberg, Germany, April 4-7, 2016, Proceedings*. Accepted for publication (cit. on p. 159).

Bruns, Felix, Dirk Kuschnerus, and Attila Bilgic (2013). "Virtualization for Safety-critical, Deeply-embedded Devices". In: *Proceedings of the 28th Annual ACM Symposium on Applied Computing*. SAC '13. Coimbra, Portugal: ACM, pp. 1485–1492. ISBN: 9781450316569. DOI: 10.1145/2480362.2480640 (cit. on p. 27).

Bui, Dai Nguyen, Edward A. Lee, Isaac Liu, Hiren D. Patel, and Jan Reineke (2011). "Temporal Isolation on Multiprocessing Architectures". In: *Design Automation Conference (DAC)*, pp. 274–279. DOI: 10.1145/2024724.2024787 (cit. on p. 4).

Burns, Alan, Divya Prasad, Andrea Bondavalli, Felicita Di Giandomenico, Krithi Ramamritham, John Stankovic, and Lorenzo Strigini (2000). "The meaning and role of value in scheduling flexible real-time systems". In: *Journal of Systems Architecture* 46.4, pp. 305–325. ISSN: 1383-7621. DOI: 10.1016/S1383-7621(99)00008-9 (cit. on p. 85).

Buttazzo, G., M. Spuri, and F. Sensini (1995). "Value vs. deadline scheduling in overload conditions". In: *16th IEEE Real-Time Systems Symposium, 1995. Proceedings.* Pp. 90–99. DOI: 10.1109/REAL.1995.495199 (cit. on p. 89).

Buttazzo, Giorgio C., Enrico Bini, and Darren Buttle (2014). "Rate-Adaptive Tasks: Model, Analysis, and Design Issues". In: *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*. Dresden, Germany, pp. 1–6. DOI: 10.7873/DATE.2014.266 (cit. on p. 73).

Buttazzo, Giorgio C., Giuseppe Lipari, Marco Caccamo, and Luca Abeni (2002). "Elastic Scheduling for Flexible Workload Management". In: *IEEE Trans. Comput.* 51.3, pp. 289–302. ISSN: 0018-9340. DOI: 10.1109/12.990127 (cit. on p. 73).

Buttazzo, Giorgio and Enrico Bini (2006). "Optimal Dimensioning of a Constant Bandwidth Server". In: *27th IEEE International Real-Time Systems Symposium, 2006. RTSS '06.* Pp. 169–177. DOI: 10.1109/RTSS.2006.31 (cit. on p. 123).

Carrascosa, E., Javier Coronel, Miguel Masmano, Patricia Balbastre, and Alfons Crespo (2014). "XtratuM Hypervisor Redesign for LEON4 Multicore Processor". In: *SIGBED Review* 11.2, pp. 27–31. ISSN: 1551-3688. DOI: 10.1145/2668138.2668142 (cit. on p. 27).

Caspi, Paul, Adrian Curic, Aude Maignan, Christos Sofronis, Stavros Tripakis, and Peter Niebert (2003). "From Simulink to SCADE/Lustre to TTA: A Layered Approach for Distributed Embedded Applications". In: *Proceedings of the 2003 ACM SIGPLAN Conference on Language, Compiler, and Tool for Embedded Systems*. LCTES '03. San Diego, California, USA: ACM, pp. 153–162. ISBN: 1581136471. DOI: 10.1145/780732.780754 (cit. on p. 120).

Chakraborty, Samarjit, Simon Künzli, and Lothar Thiele (2003). "A general framework for analysing system properties in platform-based embedded system designs". In: *Design, Automation and Test in Europe Conference and Exhibition, 2003*, pp. 190–195. DOI: 10.1109/DATE.2003.1253607 (cit. on p. 74).

Chen, DeJiu, Rolf Johansson, Henrik Lönn, Yiannis Papadopoulos, Anders Sandberg, Fredrik Törner, and Martin Törngren (2008). "Modelling Support for Design of Safety-Critical Automotive Embedded Systems". In: *Computer Safety, Reliability, and Security*. Ed. by Michael D. Harrison and Mark-Alexander Sujan. Vol. 5219. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 72–85. ISBN: 9783540876977. DOI: 10.1007/978-3-540-87698-4_9 (cit. on p. 74).

Chen, Ken and Paul Muhlethaler (1996). "A scheduling algorithm for tasks described by Time Value Function". In: *Real-Time Systems* 10 (3). 10.1007/BF00383389, pp. 293–312. ISSN: 0922-6443. DOI: 10.1007/BF00383389 (cit. on p. 89).

Chiang, Chuan-Yen, Yen-Lin Chen, Kun-Cing Ke, and Shyan-Ming Yuan (2015). "Real-time pedestrian detection technique for embedded driver assistance systems". In: *IEEE International Conference on Consumer Electronics (ICCE), 2015*, pp. 206–207. DOI: 10.1109/ICCE.2015.7066383 (cit. on p. 3).

Cho, Hyeonjoong, Yongwha Chung, and Daihee Park (2010). "Guaranteed dynamic priority assignment scheme for streams with (m, k)-firm deadlines". In: *ETRI Journal* 32.3, pp. 500–502. ISSN: 1225-6463. DOI: http://dx.doi.org/10.4218/etrij.10.0109.0544 (cit. on pp. 88, 90, 92, 101, 115).

Cho, Hyeonjoong, Binoy Ravindran, and E. Douglas Jensen (2010). "Utility accrual real-time scheduling for multiprocessor embedded systems". In: *J. Parallel Distrib. Comput.* 70.2, pp. 101–110 (cit. on p. 89).

Cho, Hyeonjoong, Haisang Wu, Binoy Ravindran, and E. Douglas Jensen (2006). "On Multiprocessor Utility Accrual Real-Time Scheduling with Statistical Timing Assurances". In: *Embedded and Ubiquitous Computing, International Conference, EUC 2006, Seoul, Korea, August 1-4, 2006, Proceedings*. Ed. by Edwin Hsing-Mean Sha, Sung-Kook Han, Cheng-Zhong Xu, Moon-hae Kim, Laurence Tianruo Yang, and Bin Xiao. Vol. 4096. Lecture Notes in Computer Science. Springer, pp. 274–286 (cit. on p. 89).

Clark, Raymond Keith (1990). "Scheduling Dependent Real-Time Activities". PhD thesis. Carnegie Mellon University (cit. on p. 89).

Clarke, Edmund M. and Jeannette M. Wing (1996). "Formal Methods: State of the Art and Future Directions". In: *ACM Computing Surveys* 28.4, pp. 626–643. ISSN: 0360-0300. DOI: 10.1145/242223.242257 (cit. on p. 8).

Colmenares, Juan A., Sarah Bird, Henry Cook, Paul Pearce, David Zhu, John Shalf, Steven Hofmeyr, Krste Asanovic, and John Kubiatowicz (2010). "Resource Management in the Tessellation Manycore OS". In: *Second USENIX Conference on Hot Topics in Parallelism (HotPAR 2010)* (cit. on pp. 21, 22).

Colmenares, Juan, Ian Saxton, Eric Battenberg, Rimas Avizienis, Nils Peters, Krste Asanovic, John D. Kubiatowicz, and David Wessel (2011). "Real-Time Musical Applications on an Experimental Operating System for Multi-Core Processors". In: *International Computer Music Conference, ICMC 2011*. Huddersfield, England (cit. on p. 22).

Craig, T.S. (1993). "Queuing spin lock algorithms to support timing predictability". In: *Real-Time Systems Symposium, 1993., Proceedings.* Pp. 148–157. DOI: 10.1109/REAL.1993.393505 (cit. on p. 46).

Crespo, Alfons, Ismael Ripoll, and Miguel Masmano (2010). "Partitioned Embedded Architecture Based on Hypervisor: The XtratuM Approach". In: *European Dependable Computing Conference (EDCC), 2010*, pp. 67–72. DOI: 10.1109/EDCC.2010.18 (cit. on pp. 3, 27).

Crippa, L., R. Micheloni, I. Motta, and M. Sangalli (2008). "Nonvolatile Memories: NOR vs. NAND Architectures". English. In: *Memories in Wireless Systems*. Ed. by Rino Micheloni, Giovanni Campardo, and Piero Olivo. Signals and Communication Technology. Springer Berlin Heidelberg, pp. 29–53. ISBN: 9783540790778. DOI: 10.1007/978-3-540-79078-5_2 (cit. on p. 61).

Cullmann, Christoph, Christian Ferdinand, Gernot Gebhard, Daniel Grund, Claire Maiza, Jan Reineke, Benoît Triquet, and Reinhard Wilhelm (2010). "Predictability Considerations in the Design of Multi-Core Embedded Systems". In: *Proceedings of Embedded Real Time Software and Systems*, pp. 36–42 (cit. on pp. 4, 13).

Dally, William J. and Brian Towles (2001). "Route packets, not wires: on-chip interconnection networks". In: *Design Automation Conference, 2001. Proceedings*, pp. 684–689. DOI: 10.1109/DAC.2001.156225 (cit. on p. 11).

D'Ausbourg, Bruno, Marc Boyer, Eric Noulard, and Claire Pagetti (2011). "Deterministic Execution on Many-Core Platforms: application to the SCC". In: *4th symposium of the Many-core Applications Research Community (MARC)* (cit. on pp. 4, 33).

David, Vincent, Jean Delcoigne, Evelyne Leret, Alain Ourghanlian, Philippe Hilsenkopf, and Philippe Paris (1998). "Safety Properties Ensured by the OASIS Model for Safety Critical Real-Time Systems". English. In: *Computer Safety, Reliability and Security*. Ed. by Wolfgang Ehrenberger. Vol. 1516. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 45–59. ISBN: 9783540651109. DOI: 10.1007/3-540-49646-7_4 (cit. on pp. 74, 121).

Davis, Robert I. and Alan Burns (2011). "A Survey of Hard Real-time Scheduling for Multiprocessor Systems". In: *ACM Computing Surveys* 43.4, 35:1–35:44. ISSN: 0360-0300. DOI: 10.1145/1978802.1978814 (cit. on p. 5).

Dinechin, Benoît Dupont de (2015). *Revisiting DSP Acceleration with the Kalray MPPA Manycore Processor*. Workshop presentation at Hot Chips 27, August 24th 2015, Cupertino, CA, USA (cit. on p. 12).

Dinechin, Benoît Dupont de, Renaud Ayrignac, Pierre-Edouard Beaucamps, Patrice Couvert, Benoît Ganne, Pierre Guironnet de Massas, François Jacquet, Samuel Jones, Nicolas Morey Chaisemartin, Frédéric Riss, and Thierry Strudel (2013). "A clustered manycore processor architecture for embedded and accelerated applications". In: *IEEE High Performance Extreme Computing Conference (HPEC), 2013*, pp. 1–6. DOI: 10.1109/HPEC.2013.6670342 (cit. on pp. 4, 12, 124).

Ding, Wanfu and Ruifeng Guo (2008). "Design and Evaluation of Sectional Real-Time Scheduling Algorithms Based on System Load". In: *The 9th International Conference for Young Computer Scientists, 2008. ICYCS 2008.* Pp. 14–18. DOI: 10.1109/ICYCS.2008.208 (cit. on p. 89).

Draves, Richard P., Brian N. Bershad, Richard F. Rashid, and Randall W. Dean (1991). "Using Continuations to Implement Thread Management and Communication in Operating Systems". In: *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*. SOSP '91. Pacific Grove, California, USA: ACM, pp. 122–136. ISBN: 0897914473. DOI: 10.1145/121132.121155 (cit. on p. 20).

Dunkels, Adam, Leon Woestenberg, Kieran Mansley, and Leon Woestenberg. *lwIP - A Lightweight TCP/IP stack*. Accessed 2012. URL: `http://savannah.nongnu.org/projects/lwip/` (cit. on pp. 16, 20).

Dunn, William R. (2003). "Designing Safety-critical Computer Systems". In: *IEEE Computer* 36.11, pp. 40–46. ISSN: 0018-9162. DOI: `10.1109/MC.2003.1244533` (cit. on p. 8).

Ekelin, Cecilia (2006). "Clairvoyant Non-Preemptive EDF Scheduling". In: *18th Euromicro Conference on Real-Time Systems, ECRTS'06, 5-7 July 2006, Dresden, Germany, Proceedings*. IEEE Computer Society, pp. 23–32. ISBN: 0769526195. DOI: `10.1109/ECRTS.2006.7` (cit. on p. 122).

Engler, D. R., M. F. Kaashoek, and J. O'Toole Jr. (1995). "Exokernel: An Operating System Architecture for Application-Level Resource Management". In: *Proceedings of the fifteenth ACM symposium on Operating systems principles*. SOSP '95. Copper Mountain, Colorado, United States: ACM, pp. 251–266. ISBN: 0897917154. DOI: `10.1145/224056.224076` (cit. on pp. 15, 18).

Espinoza, Huáscar, Hubert Dubois, Sébastien Gérard, Julio Medina, Dorina C. Petriu, and Murray Woodside (2006). "Annotating UML Models with Non-functional Properties for Quantitative Analysis". In: *Satellite Events at the MoDELS 2005 Conference*. Ed. by Jean-Michel Bruel. Vol. 3844. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 79–90. ISBN: 9783540317807. DOI: `10.1007/11663430_9` (cit. on p. 74).

Fähndrich, Manuel, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen Hunt, James R. Larus, and Steven Levi (2006). "Language support for fast and reliable message-based communication in singularity OS". In: *SIGOPS Operating Systems Review* 40 (4), pp. 177–190. ISSN: 0163-5980. DOI: `10.1145/1218063.1217953` (cit. on p. 24).

Fitzgerald, Robert, Todd B. Knoblock, Erik Ruf, Bjarne Steensgaard, and David Tarditi (1999). *Marmot: An Optimizing Compiler for Java*. Tech. rep. MSR-TR-99-33. Microsoft Research (cit. on p. 24).

Flavia, Felicioni, Jia Ning, Françoise Simonot-Lion, and Song YeQiong (2008). "Optimal online (m,k)-firm constraint assignment for real-time control tasks based on plant state information". In: *IEEE International Conference on Emerging Technologies and Factory Automation, 2008. ETFA 2008.* Pp. 908–915. DOI: `10.1109/ETFA.2008.4638504` (cit. on p. 88).

Fontanelli, Daniele, Luca Greco, and Luigi Palopoli (2013). "Soft real-time scheduling for embedded control systems". In: *Automatica* 49.8, pp. 2330–2338. ISSN: 0005-1098. DOI: `10.1016/j.automatica.2013.04.036` (cit. on p. 120).

FreeEMS. *FreeEMS – Free and Open Source Engine Management System*. http://freeems.org/ (cit. on p. 73).

Garey, Michael R. and David S. Johnson (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. San Fransisco: W.H. Freeman (cit. on p. 122).

George, Laurent, Paul Mühlethaler, and Nicolas Rivierre (2000). *A Few Results on Non-Preemptive Real-time Scheduling*. Rapport de recherche N° 3926. Institut National de Recherche en Informatique et en Automatique (cit. on p. 122).

George, Laurent, Nicolas Rivierre, and Marco Spuri (1996). *Preemptive and Non-Preemptive Real-Time Uni-Processor Scheduling*. Rapport de recherche N° 2966. Institut National de Recherche en Informatique et en Automatique (cit. on p. 52).

Gerdes, Mike, Florian Kluge, Theo Ungerer, Christine Rochange, and Pascal Sainrat (2012). "Time Analysable Synchronisation Techniques for Parallelised Hard Real-Time Applica-

tions". In: *Proc. Design, Automation and Testing in Europe (DATE 2012)*, pp. 671–676 (cit. on p. 46).

Ghosal, Arkadeb, Thomas A. Henzinger, Christoph M. Kirsch, and Marco A.A. Sanvido (2004). "Event-Driven Programming with Logical Execution Times". English. In: *Hybrid Systems: Computation and Control*. Ed. by Rajeev Alur and GeorgeJ. Pappas. Vol. 2993. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 357–371. ISBN: 9783540212591. DOI: 10.1007/978-3-540-24743-2_24 (cit. on p. 120).

Ghosal, Arkadeb, Alberto Sangiovanni-Vincentelli, Christoph M. Kirsch, Thomas A. Henzinger, and Daniel Iercan (2006). "A Hierarchical Coordination Language for Interacting Real-time Tasks". In: *Proceedings of the 6th ACM & IEEE International Conference on Embedded Software*. EMSOFT '06. Seoul, Korea: ACM, pp. 132–141. ISBN: 1595935428. DOI: 10.1145/1176887.1176907 (cit. on p. 120).

Goossens, Joël (2008). "$(m, k)$-Firm Constraints and DBP Scheduling: Impact of the Initial k-Sequence and Exact Feasibility Test". In: *16th International Conference on Real-Time and Network Systems (RTNS'08)*, pp. 61–66 (cit. on pp. 88, 90, 93, 98, 100, 102, 106, 108, 115).

Goossens, Joël and Christophe Macq (2001). "Limitation of the Hyper-Period in Real-Time Periodic Task Set Generation". In: *Proceedings of the 9th International Conference on Real-Time Systems (RTS'01)*, pp. 133–148 (cit. on pp. 100, 102, 113–115).

Goossens, Kees, John Dielissen, and Andrei Radulescu (2005). "Æthereal Network on Chip: Concepts, Architectures, and Implementations". In: *IEEE Design & Test of Computers* 22.5, pp. 414–421. ISSN: 0740-7475. DOI: 10.1109/MDT.2005.99 (cit. on pp. 32, 70).

Green Hills. *Real-Time Operating Systems (RTOS), Embedded Development Tools, Optimizing Compilers, IDE tools, Debuggers*. http://www.ghs.com. Green Hills Software (cit. on p. 27).

Gresser, Klaus (1993). "An Event Model for Deadline Verification of Hard Real-Time Systems". In: *Fifth Euromicro Workshop on Real-Time Systems, 1993*. Pp. 118–123. DOI: 10.1109/EMWRT.1993.639067 (cit. on p. 74).

Gustafsson, Jan, Adam Betts, Andreas Ermedahl, and Björn Lisper (2010). "The Mälardalen WCET Benchmarks: Past, Present And Future". In: *10th International Workshop on Worst-Case Execution Time Analysis, WCET 2010, July 6, 2010, Brussels, Belgium*, pp. 136–146. DOI: 10.4230/OASIcs.WCET.2010.136 (cit. on p. 124).

Gut, Georg and Christian Allmann (2012). "Research Project E-Performance - In-Car-Network Optimization for Electric Vehicles". English. In: *Advanced Microsystems for Automotive Applications 2012*. Ed. by Gereon Meyer. Springer Berlin Heidelberg, pp. 69–78. ISBN: 9783642296727. DOI: 10.1007/978-3-642-29673-4_7 (cit. on p. 3).

Halbwachs, Nicholas, Paul Caspi, Pascal Raymond, and Daniel Pilaud (1991). "The synchronous data flow programming language LUSTRE". In: *Proceedings of the IEEE* 79.9, pp. 1305–1320. ISSN: 0018-9219. DOI: 10.1109/5.97300 (cit. on p. 120).

Halbwachs, Nicolas (1992). *Synchronous Programming of Reactive Systems*. Norwell, MA, USA: Kluwer Academic Publishers. ISBN: 0792393112 (cit. on p. 120).

Hamdaoui, Moncef and Parameswaran Ramanathan (1995). "A dynamic priority assignment technique for streams with (m, k)-firm deadlines". In: *IEEE Transactions on Computers* 44.12, pp. 1443–1451. ISSN: 0018-9340. DOI: 10.1109/12.477249 (cit. on pp. 82, 87, 88, 90, 101, 115).

Hanbo, Wang, Zhou Xingshe, Dong Yunwei, and Tang Lei (2009). "Modeling Timing Behavior for Cyber-Physical Systems". In: *International Conference on Computational Intelligence*

*and Software Engineering, 2009. CiSE 2009.* Pp. 1–4. DOI: 10.1109/CISE.2009.5364357 (cit. on p. 74).

Hand, Steven, Andrew Warfield, Keir Fraser, Evangelos Kotsovinos, and Dan Magenheimer (2005). "Are Virtual Machine Monitors Microkernels Done Right?" In: *Proceedings of the 10th Conference on Hot Topics in Operating Systems - Volume 10*. HOTOS'05. Santa Fe, NM: USENIX Association, pp. 1–1 (cit. on p. 27).

Hansen, Michael R. and Zhou Chaochen (1997). "Duration calculus: Logical foundations". English. In: *Formal Aspects of Computing* 9.3, pp. 283–330. ISSN: 0934-5043. DOI: 10.1007/BF01211086 (cit. on p. 74).

Heiser, Gernot (2008). "The Role of Virtualization in Embedded Systems". In: *Proceedings of the 1st Workshop on Isolation and Integration in Embedded Systems*. IIES '08. Glasgow, Scotland: ACM, pp. 11–16. ISBN: 9781605581262. DOI: 10.1145/1435458.1435461 (cit. on pp. 3, 10, 27).

— (2011). "Virtualizing Embedded Systems: Why Bother?" In: *Proceedings of the 48th Design Automation Conference*. DAC '11. San Diego, California: ACM, pp. 901–905. ISBN: 9781450306362. DOI: 10.1145/2024724.2024925 (cit. on pp. 10, 27).

Heiser, Gernot, Volkmar Uhlig, and Joshua LeVasseur (2006). "Are Virtual-machine Monitors Microkernels Done Right?" In: *SIGOPS Operating Systems Review* 40.1, pp. 95–99. ISSN: 0163-5980. DOI: 10.1145/1113361.1113363 (cit. on p. 27).

Hemani, Ahmed, Axel Jantsch, Shashi Kumar, Adam Postula, Johnny Oberg, Mikael Millberg, and Dan Lindqvist (2000). "Network on chip: An architecture for billion transistor era". In: *Proceeding of the IEEE NorChip Conference*. Vol. 31 (cit. on p. 11).

Henzinger, Thomas A., Benjamin Horowitz, and Christoph M. Kirsch (2003). "Giotto: a time-triggered language for embedded programming". In: *Proceedings of the IEEE* 91.1, pp. 84–99. ISSN: 0018-9219. DOI: 10.1109/JPROC.2002.805825 (cit. on pp. 74, 120, 121).

Hillenbrand, Martin (2012). "Funktionale Sicherheit nach ISO 26262 in der Konzeptphase der Entwicklung von Elektrik/Elektronik Architekturen von Fahrzeugen". PhD thesis. Karlsruher Institut für Technologie, Institut für Technik der Informationsverarbeitung. ISBN: 9783866448032 (cit. on p. 7).

Hoare, Charles Anthony Richard (1969). "An Axiomatic Basis for Computer Programming". In: *Commununications of the ACM* 12.10, pp. 576–580. ISSN: 0001-0782. DOI: 10.1145/363235.363259 (cit. on p. 8).

— (1978). "Communicating sequential processes". In: *Communications of the ACM* 21.8, pp. 666–677. ISSN: 0001-0782. DOI: 10.1145/359576.359585 (cit. on pp. 17, 52).

Hong, Shengyan, Thidapat Chantem, and Xiaobo Sharon Hu (2011). "Meeting End-to-End Deadlines through Distributed Local Deadline Assignments". In: *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*, pp. 183–192. DOI: 10.1109/RTSS.2011.24 (cit. on p. 122).

Howard, Jason, Saurabh Dighe, Yatin Hoskote, Sriram R. Vangal, David Finan, Gregory Ruhl, David Jenkins, Howard Wilson, Nitin Borkar, Gerhard Schrom, Fabric Pailet, Shailendra Jain, Tiju Jacob, Satish Yada, Sraven Marella, Praveen Salihundam, Vasantha Erraguntla, Michael Konow, Michael Riepen, Guido Droege, Joerg Lindemann, Matthias Gries, Thomas Apel, Kersten Henriss, Tor Lund-Larsen, Sebastian Steibl, Shekhar Borkar, Vivek De, Rob F. Van der Wijngaart, and Timothy G. Mattson (2010). "A 48-Core IA-32 message-passing processor with DVFS in 45nm CMOS". In: *IEEE International Solid-State Circuits Conference, ISSCC 2010, Digest of Technical Papers, San Francisco, CA,*

*USA, 7-11 February, 2010*. IEEE, pp. 108–109. ISBN: 9781424460335. DOI: 10.1109/ISSCC.2010.5434077 (cit. on p. 4).

Howell, Rodney R. and Muralidhar K. Venkatrao (1995). "On Non-preemptive Scheduling of Recurring Tasks Using Inserted Idle Times". In: *Information and Computation* 117.1, pp. 50–62. ISSN: 0890-5401. DOI: 10.1006/inco.1995.1028 (cit. on p. 122).

Hunt, Galen C., James R. Larus, David Tarditi, and Ted Wobber (2005). "Broad new OS research: challenges and opportunities". In: *Proceedings of the 10th conference on Hot Topics in Operating Systems - Volume 10*. HOTOS'05. Santa Fe, NM: USENIX Association, pp. 15–15 (cit. on p. 24).

Infineon (2014). *Highly Integrated and Performance Optimized 32-bit Microcontrollers for Automotive and Industrial Applications*. Neubiberg, Germany: Infineon Technologies AG (cit. on pp. 4, 12).

Intel (2013). *Intel Xeon Phi Coprocessor System Software Developers Guide*. SKU#328207-002EN. Intel Corporation (cit. on p. 55).

— (2014). *Intel Xeon Phi Coprocessor, Datasheet*. Intel Corporation (cit. on pp. 4, 11).

Jensen, E. Douglas, C. Douglas Locke, and Hideyuki Tokuda (1985). "A Time-Driven Scheduling Model for Real-Time Operating Systems". In: *6th Real-Time Systems Symposium (RTSS '85), December 3-6, 1985, San Diego, California, USA*, pp. 112–122 (cit. on pp. 5, 80, 87, 89, 97).

Jia, Ning, Emmanuel Hyon, and Ye-Qiong Song (2005). "Ordonnancement sous contraintes (m,k)-firm et combinatoire des mots". In: *13th International Conference on Real-Time Systems - RTS'2005*. Nicolas Navet. Paris/France: BIRP (cit. on p. 88).

Jia, Ning, Ye-Qiong Song, and Françoise Simonot-Lion (2007). "Task Handler Based on (m,k)-firm Constraint Model for Managing a Set of Real-Time Controllers". In: *15th International Conference on Real-Time and Network Systems - RTNS 2007*. Ed. by Nicolas Navet, Françoise Simonot-Lion, and Isabelle Puaut. Nancy, France, pp. 183–194 (cit. on pp. 88, 90, 92–94, 100, 101, 106, 114).

Jonsson, Jan and Kang G. Shin (1997). "Deadline assignment in distributed hard real-time systems with relaxed locality constraints". In: *Proceedings of the 17th International Conference on Distributed Computing Systems, 1997*. Pp. 432–440. DOI: 10.1109/ICDCS.1997.598077 (cit. on p. 122).

Jordan, Alexander, Florian Brandner, and Martin Schoeberl (2013). "Static Analysis of Worst-case Stack Cache Behavior". In: *Proceedings of the 21st International Conference on Real-Time Networks and Systems*. RTNS '13. Sophia Antipolis, France: ACM, pp. 55–64. ISBN: 9781450320580. DOI: 10.1145/2516821.2516828 (cit. on p. 47).

Kaiser, Robert and Stephan Wagner (2007). "Evolution of the PikeOS microkernel". In: *Proceedings of the 1st International Workshop on Microkernels for Embedded Systems*, pp. 50–57 (cit. on p. 27).

Kasapaki, Evangelia, Jens Sparsø, Rasmus Bo Sorensen, and Kees Goossens (2013). "Router Designs for an Asynchronous Time-Division-Multiplexed Network-on-Chip". In: *2013 Euromicro Conference on Digital System Design, DSD 2013, Los Alamitos, CA, USA, September 4-6, 2013*. IEEE Computer Society, pp. 319–326. ISBN: 9781479929788. DOI: 10.1109/DSD.2013.40 (cit. on p. 47).

Kesting, Arne, Martin Treiber, Martin Schönhof, and Dirk Helbing (2007). "Extending Adaptive Cruise Control to Adaptive Driving Strategies". In: *Transportation Research Record: Journal of the Transportation Research Board* 2000, pp. 16–24. DOI: 10.3141/2000-03. eprint: http://dx.doi.org/10.3141/2000-03 (cit. on p. 3).

Kim, Albert, Juan A. Colmenares, Hilfi Alkaff, and John Kubiatowicz (2012). "A Soft Real-Time, Parallel GUI Service in Tessellation Many-Core OS". In: *27th International Conference on Computers and Their Applications (CATA 2012)* (cit. on p. 22).

Kinnan, Larry M. (2009). "Use of multicore processors in avionics systems and its potential impact on implementation and certification". In: *28th IEEE/AIAA Digital Avionics Systems Conference, 2009 (DASC '09)*, 1.E.4-1–1.E.4-6. DOI: 10.1109/DASC.2009.5347560 (cit. on pp. 4, 13, 14).

Kirsch, Christoph M. (2002). "Principles of Real-Time Programming". In: *Embedded Software, Second International Conference, EMSOFT 2002, Grenoble, France, October 7-9, 2002, Proceedings*, pp. 61–75. DOI: 10.1007/3-540-45828-X_6 (cit. on p. 119).

Kirsch, Christoph M. and Raja Sengupta (2007). "The Evolution of Real-Time Programming". In: *Handbook of Real-Time and Embedded Systems*. Ed. by Insup Lee, Joseph Y-T. Leung, and Sang H. Son. 1st ed. Chapman & Hall/CRC. DOI: 10.1201/9781420011746.ch11 (cit. on pp. 119, 120).

Klues, Kevin, Barret Rhoden, Andrew Waterman, David Zhu, and Eric Brewer (2010). "Processes and Resource Management in a Scalable Many-core OS". In: *HotPar10*. Berkeley, CA (cit. on p. 23).

Kluge, Florian (2014). *tms-sim – Timing Models Scheduling Simulation Framework – Release 2014-12*. Tech. rep. 2014-07. Department of Computer Science, University of Augsburg. DOI: 10.13140/2.1.1251.2321 (cit. on pp. 99, 159).

— (2015). *A Simple Capture/Compare Timer*. Tech. rep. 2015-01. Department of Computer Science, University of Augsburg. DOI: 10.13140/2.1.1251.2321 (cit. on p. 160).

— (2016). *Notes on the Generation of Spin-Values for Fixed $(m, k)$-Patterns*. Tech. rep. 2016-01. Department of Computer Science, University of Augsburg (cit. on p. 92).

Kluge, Florian, Mike Gerdes, Florian Haas, and Theo Ungerer (2014). "A Generic Timing Model for Cyber-Physical Systems". In: *Workshop Reconciling Performance and Predictability (RePP'14)*. Grenoble, France. DOI: 10.13140/2.1.1820.4165 (cit. on pp. 73, 74).

Kluge, Florian, Mike Gerdes, and Theo Ungerer (2012). "AUTOSAR OS on a Message-Passing Multicore Processor". In: *Proceedings of 7th IEEE International Symposium on Industrial Embedded Systems (SIES'12) (Posters)*. Karlsruhe, Germany (cit. on p. 42).

— (2014a). "An Operating System for Safety-Critical Applications on Manycore Processors". In: *17th IEEE International Symposium on Object Oriented Real-Time Distributed Computing (ISORC), 2014*. Reno, NV, USA: IEEE, pp. 238–245 (cit. on pp. 31, 36).

— (2014b). "The Boot Process in Real-time Manycore Processors". In: *Proceedings of the 22nd International Conference on Real-Time Networks and Systems*. RTNS '14. Versaille, France: ACM, 77:77–77:86. ISBN: 9781450327275. DOI: 10.1145/2659787.2659816 (cit. on pp. 55, 159).

Kluge, Florian, Florian Haas, Mike Gerdes, and Theo Ungerer (2013). "History-Cognisant Time-Utility-Functions for Scheduling Overloaded Real-Time Control Systems". In: *Proceedings of 7th Junior Researcher Workshop on Real-Time Computing (JRWRTC 2013)*. Sophia Antipolis, France. DOI: 10.13140/2.1.1080.0648 (cit. on pp. 88, 159).

Kluge, Florian, Markus Neuerburg, and Theo Ungerer (2015). "Utility-based scheduling of $(m, k)$-Firm Real-Time Task Sets". In: *Accepted at 28th GI/ITG International Conference on Architecture of Computing Systems (ARCS 2015)* (cit. on pp. 87, 101–103).

Kluge, Florian, Benoît Triquet, Christine Rochange, and Theo Ungerer (2012). "Operating Systems for Manycore Processors from the Perspective of Safety-Critical Systems". In:

*Proceedings of 8th annual workshop on Operating Systems for Embedded Real-Time applications (OSPERT 2012)*. Pisa, Italy. DOI: 10.13140/2.1.1211.1362 (cit. on pp. 10, 31, 36).

Kluge, Florian and Theo Ungerer (2015). "EMSBench: Benchmark und Testumgebung für reaktive Systeme". In: *Betriebssysteme und Echtzeit, Echtzeit 2015, Fachtagung des gemeinsamen Fachausschusses Echtzeitsysteme von Gesellschaft für Informatik e.V. (GI), VDI/VDE-Gesellschaft für Mess- und Automatisierungstechnik (GMA) und Informationstechnischer Gesellschaft im VDE (ITG) sowie der Fachgruppe Betriebssysteme von GI und ITG, Boppard, Germany, 12.-13. November 2015*. Ed. by Wolfgang A. Halang and Olaf Spinczyk. Informatik Aktuell. Springer, pp. 11–20. DOI: 10.1007/978-3-662-48611-5_2 (cit. on pp. 73, 124, 159).

Kluge, Florian, Chenglong Yu, Jörg Mische, Sascha Uhrig, and Theo Ungerer (2009). "Implementing AUTOSAR Scheduling and Resource Management on an Embedded SMT Processor". In: *12th International Workshop on Software and Compilers for Embedded Systems (SCOPES 2009)*. Nice, France, pp. 33–41 (cit. on p. 45).

Kong, Yeonhwa and Hyeonjoong Cho (2011). "Guaranteed Scheduling for (m,k)-firm Deadline-Constrained Real-Time Tasks on Multiprocessors". In: *Parallel and Distributed Computing, Applications and Technologies (PDCAT), 2011 12th International Conference on*, pp. 18–23. DOI: 10.1109/PDCAT.2011.44 (cit. on p. 89).

Kongetira, Poonacha, Kathirgamar Aingaran, and Kunle Olukotun (2005). "Niagara: a 32-way multithreaded Sparc processor". In: *IEEE Micro* 25.2, pp. 21–29. ISSN: 0272-1732. DOI: 10.1109/MM.2005.35 (cit. on p. 11).

Koren, Gilad and Dennis Shasha (1992). "Dover; an optimal on-line scheduling algorithm for overloaded real-time systems". In: *Real-Time Systems Symposium, 1992*, pp. 290–299. DOI: 10.1109/REAL.1992.242650 (cit. on p. 89).

— (1995). "Skip-Over: algorithms and complexity for overloaded systems that allow skips". In: *16th IEEE Real-Time Systems Symposium, 1995. Proceedings.* Pp. 110–117. DOI: 10.1109/REAL.1995.495201 (cit. on p. 87).

Kramer, Simon, Dirk Ziegenbein, and Arne Hamann (2015). "Real world automotive benchmark for free". In: *6th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS 2015), July 7, 2015, Lund, Sweden*. Lund, Sweden (cit. on p. 111).

Le Lann, Gérard (1997). "An analysis of the Ariane 5 flight 501 failure-a system engineering perspective". In: *International Conference and Workshop on Engineering of Computer-Based Systems, 1997. Proceedings.*, pp. 339–346. DOI: 10.1109/ECBS.1997.581900 (cit. on p. 7).

Lee, Edward A. (2008). *Cyber Physical Systems: Design Challenges*. Tech. rep. UCB/EECS-2008-8. EECS Department, University of California, Berkeley (cit. on p. 73).

Lehoczky, John, Lui Sha, and Ye Ding (1989). "The rate monotonic scheduling algorithm: exact characterization and average case behavior". In: *Real Time Systems Symposium, 1989., Proceedings.* Pp. 166–171. DOI: 10.1109/REAL.1989.63567 (cit. on pp. 94, 100, 115).

Leveson, Nancy G. (1986). "Software Safety: Why, What, and How". In: *ACM Computing Surveys* 18.2, pp. 125–163. ISSN: 0360-0300. DOI: 10.1145/7474.7528 (cit. on p. 7).

Leveson, N.G. and C.S. Turner (1993). "An investigation of the Therac-25 accidents". In: *Computer* 26.7, pp. 18–41. ISSN: 0018-9162. DOI: 10.1109/MC.1993.274940 (cit. on p. 7).

Li, Jian, Yeqiong Song, and Françoise Simonot-Lion (2004). "Schedulability analysis for systems under (m,k)-firm constraints". In: *2004 IEEE International Workshop on Factory Communication Systems, 2004. Proceedings.* Pp. 23–30. DOI: 10.1109/WFCS.2004.1377670 (cit. on p. 88).

Li, Peng and Binoy Ravindran (2004). "Fast, best-effort real-time scheduling algorithms". In: *IEEE Transactions on Computers* 53.9, pp. 1159–1175. ISSN: 0018-9340. DOI: 10.1109/TC.2004.61 (cit. on p. 89).

Li, Peng, Haisang Wu, Binoy Ravindran, and E. Douglas Jensen (2006). "A utility accrual scheduling algorithm for real-time activities with mutual exclusion resource constraints". In: *IEEE Transactions on Computers* 55.4, pp. 454–469. ISSN: 0018-9340. DOI: 10.1109/TC.2006.47 (cit. on p. 89).

Liedtke, Jochen (1995). "On micro-kernel construction". In: *Proceedings of the fifteenth ACM symposium on Operating systems principles*. SOSP '95. Copper Mountain, Colorado, United States: ACM, pp. 237–250. ISBN: 0897917154. DOI: 10.1145/224056.224075 (cit. on pp. 16, 27).

Lim, SungKyu (2013). "3D-MAPS: 3D Massively Parallel Processor with Stacked Memory". English. In: *Design for High Performance, Low Power, and Reliable 3D Integrated Circuits*. Springer New York, pp. 537–560. ISBN: 9781441995414. DOI: 10.1007/978-1-4419-9542-1_20 (cit. on p. 13).

Lions, Jacques-Louis (1996). *ARIANE 5 Flight 501 Failure: Report by the Inquiry Board*. Report by the Inquiry Board. http://esamultimedia.esa.int/docs/esa-x-1819eng.pdf, retrieved 18.06.2015 (cit. on p. 7).

Liu, Chung Laung and James W. Layland (1973). "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment". In: *Journal of the ACM* 20.1, pp. 46–61. ISSN: 0004-5411. DOI: 10.1145/321738.321743 (cit. on pp. 5, 53, 73, 91, 92, 113, 122).

Liu, Rose, Kevin Klues, Sarah Bird, Steven Hofmeyr, Krste Asanović, and John Kubiat (2009). "Tessellation: Space-Time Partitioning in a Manycore Client OS". In: *Proc. 1st Workshop on Hot Topics in Parallelism (Hot-Par '09)* (cit. on p. 21).

Locke, Carey Douglass (1986). "Best-effort decision-making for real-time scheduling". PhD thesis. Pittsburgh, PA, USA: Carnegie Mellon University (cit. on pp. 87, 89, 97).

Loh, Gabriel H. (2008). "3D-Stacked Memory Architectures for Multi-core Processors". In: *Proceedings of the 35th Annual International Symposium on Computer Architecture*. ISCA '08. Washington, DC, USA: IEEE Computer Society, pp. 453–464. ISBN: 9780769531748. DOI: 10.1109/ISCA.2008.15 (cit. on pp. 13, 32).

Lynx. *LynxSecure Separation Kernel Hypervisor*. http://www.lynx.com/products/secure-virtualization/lynxsecure-separation-kernel-hypervisor/. Lynx Software Technologies, Inc. (cit. on p. 27).

Masmano, Miguel, Ismael Ripoll, Alfons Crespo, and Jean-Jacques Metge (2009). "Xtratum: a hypervisor for safety critical embedded systems". In: *11th Real-Time Linux Workshop*. Citeseer, pp. 263–272 (cit. on p. 27).

Mejía-Alvarez, Pedro, Rami Melhem, and Daniel Mossé (2000). "An incremental approach to scheduling during overloads in real-time systems". In: *The 21st IEEE Real-Time Systems Symposium, 2000. Proceedings.* Pp. 283–293. DOI: 10.1109/REAL.2000.896017 (cit. on p. 89).

Metzlaff, Stefan, Jörg Mische, and Theo Ungerer (2011). "A Real-Time Capable Many-Core Model". In: *Work-in-Progress Session of the 32nd IEEE Real-Time Systems Symposium (RTSS 2011)*. Vienna, Austria (cit. on pp. 31, 37, 57, 126).

Metzlaff, Stefan and Theo Ungerer (2014). "A comparison of instruction memories from the WCET perspective". In: *Journal of Systems Architecture - Embedded Systems Design* 60.5, pp. 452–466. DOI: 10.1016/j.sysarc.2013.09.009 (cit. on p. 32).

Microsoft Corporation (2008). *Singularity RDK*. http://singularity.codeplex.com/ (cit. on p. 24).

Milner, Robin, Joachim Parrow, and David Walker (1992). "A calculus of mobile processes, I". In: *Information and Computation* 100.1, pp. 1–40. ISSN: 0890-5401. DOI: 10.1016/0890-5401(92)90008-4 (cit. on p. 17).

Mita, Tsutomu (1985). "Optimal digital feedback control systems counting computation time of control laws". In: *Automatic Control, IEEE Transactions on* 30.6, pp. 542–548. ISSN: 0018-9286. DOI: 10.1109/TAC.1985.1104007 (cit. on p. 120).

Mok, Aloysius Ka-Lau (1983). "Fundamental Design Problems of Distributed Systems for The Hard-Real-Time Environment". Available as technical report No. MIT/LCS/TR-297. PhD thesis. Cambridge, MA, USA: Massachusetts Institute of Technology (cit. on p. 73).

Moore, Gordon E. (1965). "Cramming more components onto integrated circuits". In: *Electronics* 38.8, pp. 114–117 (cit. on p. 4).

Mosse, Daniel, Martha E. Pollack, and Yagíl Ronen (1999). "Value-density algorithms to handle transient overloads in scheduling". In: *Proceedings of the 11th Euromicro Conference on Real-Time Systems, 1999.* Pp. 278–286. DOI: 10.1109/EMRTS.1999.777475 (cit. on p. 89).

MPI (2012). *MPI: A Message-Passing Interface Standard*. 3.0. Message Passing Interface Forum (cit. on p. 37).

Nemer, Fadia, Hugues Cassé, Pascal Sainrat, Jean-Paul Bahsoun, and Marianne De Michiel (2006). "PapaBench: a Free Real-Time Benchmark". In: *6th International Workshop on Worst-Case Execution Time Analysis (WCET'06)*. Ed. by Frank Mueller. Vol. 4. OpenAccess Series in Informatics (OASIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN: 9783939897033. DOI: 10.4230/OASIcs.WCET.2006.678 (cit. on p. 124).

Nightingale, Edmund B., Orion Hodson, Ross McIlroy, Chris Hawblitzel, and Galen Hunt (2009). "Helios: heterogeneous multiprocessing with satellite kernels". In: *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. SOSP '09. Big Sky, Montana, USA: ACM, pp. 221–234. ISBN: 9781605587523. DOI: 10.1145/1629575.1629597 (cit. on p. 23).

Nowotsch, Jan and Michael Paulitsch (2012). "Leveraging Multi-Core Computing Architectures in Avionics". In: *2012 Ninth European Dependable Computing Conference, Sibiu, Romania, May 8-11, 2012*. IEEE Computer Society, pp. 132–143. DOI: 10.1109/EDCC.2012.27 (cit. on pp. 4, 12).

Obermaisser, R., C. El-Salloum, B. Huber, and Hermann Kopetz (2009). "From a Federated to an Integrated Automotive Architecture". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 28.7, pp. 956–965. ISSN: 0278-0070. DOI: 10.1109/TCAD.2009.2014005 (cit. on pp. 3, 9).

OSEK Group (2005). *OSEK/VDX Operating System*. Version 2.2.3. OSEK Group (cit. on p. 42).

Pagetti, Claire, Pierre Bieber, Julien Brunel, Kushal Gupta, Eric Noulard, Thierry Planche, Francois Vialard, Clément Ketchedji, Bernard Bésinet, and Philippe Despres (2012). "Re-

configurable IMA platform: from safety assessment to test scenarios on the SCARLETT demonstrator". In: *Embedded Real Time Software (ERTS'12)* (cit. on pp. 10, 28).

Pan, Heidi, Benjamin Hindman, and Krste Asanović (2009). "Lithe: Enabling Efficient Composition of Parallel Libraries". In: *Proc. 1st Workshop on Hot Topics in Parallelism (Hot-Par '09)* (cit. on p. 22).

— (2010). "Composing parallel software efficiently with lithe". In: *Proceedings of the 2010 ACM SIGPLAN conference on Programming lang uage design and implementation*. PLDI '10. Toronto, Ontario, Canada: ACM, pp. 376–387. ISBN: 9781450300193. DOI: 10.1145/1806596.1806639 (cit. on p. 22).

Pellizzoni, Rodolfo, Emiliano Betti, Stanley Bak, Gang Yao, John Criswell, Marco Caccamo, and Russell Kegley (2011). "A Predictable Execution Model for COTS-Based Embedded Systems". In: *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2011 17th IEEE*, pp. 269–279. DOI: 10.1109/RTAS.2011.33 (cit. on p. 12).

Peraldi-Frati, Marie-Agnés, Arda Goknil, Julien DeAntoni, and Johan Nordlander (2012). "A Timing Model for Specifying Multi Clock Automotive Systems: The Timing Augmented Description Language V2". In: *17th IEEE International Conference on Engineering of Complex Computer Systems, ICECCS 2012, Paris, France, July 18-20, 2012*. Ed. by Isabelle Perseil, Karin Breitman, and Marc Pouzet. IEEE Computer Society, pp. 230–239. DOI: 10.1109/ICECCS.2012.5 (cit. on p. 74).

Peter, Simon, Adrian Schüpbach, Paul Barham, Andrew Baumann, Rebecca Isaacs, Tim Harris, and Timothy Roscoe (2010). "Design principles for end-to-end multicore schedulers". In: *2nd USENIX conference on Hot topics in parallelism*. HotPar'10. Berkeley, CA, USA: USENIX Association, pp. 10–10 (cit. on p. 17).

Peter, Simon, Adrian Schüpbach, Dominik Menzi, and Timothy Roscoe (2011). "Early experience with the Barrelfish OS and the Single-Chip Cloud Computer". In: *Proceedings of the 3rd Intel Multicore Applications Research Community Symposium (MARC)*. KIT Scientific Publishing, Karlsruhe, pp. 35–39 (cit. on p. 18).

Peters, Ted (2008). *SMP Boot Process for Dual E500 Cores*. Application Note AN3542. Freescale Semiconductor (cit. on p. 55).

Poggi, Enrico, Ye-Qiong Song, Anis Koubaa, and Zhi Wang (2003). "Matrix-DBP for (m, k)-firm real-time guarantee". In: *Real-Time Systems Conference RTS'2003, pp457-482, Paris (France)* (cit. on pp. 88, 93).

Popek, Gerald J. and Robert P. Goldberg (1974). "Formal Requirements for Virtualizable Third Generation Architectures". In: *Commun. ACM* 17.7, pp. 412–421. ISSN: 0001-0782. DOI: 10.1145/361011.361073 (cit. on pp. 3, 27).

Potop-Butucaru, Dumitru, Robert de Simone, and Jean-Pierre Talpin (2005). "The Synchronous hypothesis and Synchronous Languages". In: *Embedded Systems Handbook*. Ed. by Richard Zurawski. CRC Press, pp. 8-1–8-23. ISBN: 9780849328244. DOI: 10.1201/9781420038163.ch8 (cit. on p. 120).

Pree, Wolfgang and Josef Templ (2004). "Towards a Component Architecture for Hard Real Time Control Applications". In: *Automotive Software - Connected Services in Mobile Networks, First Automotive Software Workshop, ASWSD 2004, San Diego, CA, USA, January 10-12, 2004, Revised Selected Papers*. Ed. by Manfred Broy, Ingolf H. Krüger, and Michael Meisinger. Springer, pp. 74–85. DOI: 10.1007/11823063_6 (cit. on p. 74).

— (2006). "Modeling with the Timing Definition Language (TDL)". In: *Model-Driven Development of Reliable Automotive Services, Second Automotive Software Workshop, ASWSD 2006, San Diego, CA, USA, March 15-17, 2006, Revised Selected Papers*. Ed. by Manfred

Broy, Ingolf H. Krüger, and Michael Meisinger. Vol. 4922. Lecture Notes in Computer Science. Springer, pp. 133–144. ISBN: 9783540709299. DOI: 10.1007/978-3-540-70930-5_9 (cit. on p. 120).

Prisaznuk, Paul J. (1992). "Integrated modular avionics". In: *Aerospace and Electronics Conference, 1992. NAECON 1992., Proceedings of the IEEE 1992 National*, 39–45 vol.1. DOI: 10.1109/NAECON.1992.220669 (cit. on pp. 3, 9).

Puschner, Peter P., Daniel Prokesch, Benedikt Huber, Jens Knoop, Stefan Hepp, and Gernot Gebhard (2013). "The T-CREST approach of compiler and WCET-analysis integration". In: *16th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing, ISORC 2013, Paderborn, Germany, June 19-21, 2013*. IEEE Computer Society, pp. 1–8. DOI: 10.1109/ISORC.2013.6913220 (cit. on p. 47).

Quan, Gang and Xiaobo Hu (2000). "Enhanced fixed-priority scheduling with (m,k)-firm guarantee". In: *The 21st IEEE Real-Time Systems Symposium, 2000. Proceedings.* Pp. 79–88. DOI: 10.1109/REAL.2000.895998 (cit. on pp. 88, 90, 92, 93, 101, 103, 115).

Ramanathan, Parameswaran (1999). "Overload management in real-time control applications using (m, k)-firm guarantee". In: *IEEE Transactions on Parallel and Distributed Systems* 10.6, pp. 549–559. ISSN: 1045-9219. DOI: 10.1109/71.774906 (cit. on pp. 78, 88, 90–93, 101, 115, 123).

Ravindran, Binoy, E. Douglas Jensen, and Peng Li (2005). "On recent advances in time/utility function real-time scheduling and resource management". In: *Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, 2005. ISORC 2005.* Pp. 55–60. DOI: 10.1109/ISORC.2005.39 (cit. on p. 80).

Rhoden, Barret, Kevin Klues, David Zhu, and Eric Brewer (2011). "Improving per-node efficiency in the datacenter with new OS abstractions". In: *Proceedings of the 2nd ACM Symposium on Cloud Computing*. SOCC '11. Cascais, Portugal: ACM, 25:1–25:8. ISBN: 9781450309769. DOI: 10.1145/2038916.2038941 (cit. on p. 23).

Rhu, J.-H., J.-H. Sun, K. Kim, H. Cho, and J.K. Park (2011). "Utility accrual real-time scheduling for (m, k)-firm deadline-constrained streams on multiprocessors". In: *Electronics Letters* 47.5, pp. 316–317. ISSN: 0013-5194. DOI: 10.1049/el.2010.7980 (cit. on pp. 88–90, 92, 101, 115).

Rushby, John (1999). *Partitioning for Safety and Security: Requirements, Mechanisms, and Assurance*. NASA Contractor Report CR-1999-209347. Also to be issued by the FAA. NASA Langley Research Center (cit. on pp. 3, 9).

Sacha, Jan and Sape Mullender (2012). "Networking in Osprey". In: *7th International Workshop on Plan 9, Dublin, Ireland*. Bell Labs Ireland, pp. 14–21 (cit. on p. 26).

Sacha, Jan, Jeff Napper, Sape Mullender, and Jim McKie (2012). "Osprey: Operating System for Predictable Clouds". In: *IEEE/IFIP 42nd International Conference on Dependable Systems and Networks Workshops (DSN-W), 2012*, pp. 1–6. DOI: 10.1109/DSNW.2012.6264689 (cit. on p. 25).

Sacha, Jan, Henning Schild, Jeff Napper, Noah Evans, and Sape Mullender (2013). "Message Passing and Scheduling in Osprey". In: *The 3rd Workshop on Systems for Future Multicore Architectures (SFMA'13)*. Prague, Czech Republic (cit. on p. 26).

Saidi, Selma, Rolf Ernst, Sascha Uhrig, Henrik Theiling, and Benoît Dupont de Dinechin (2015). "The Shift to Multicores in Real-time and Safety-critical Systems". In: *Proceedings of the 10th International Conference on Hardware/Software Codesign and System Synthesis*. CODES '15. Amsterdam, The Netherlands: IEEE Press, pp. 220–229. ISBN: 9781467383219. DOI: 10.1109/CODESISSS.2015.7331385 (cit. on p. 12).

Salloum, Christian El, Martin Elshuber, Oliver Höftberger, Haris Isakovic, and Armin Wasicek (2012). "The ACROSS MPSoC – A New Generation of Multi-core Processors Designed for Safety-Critical Embedded Systems". In: *Digital System Design (DSD), 2012 15th Euromicro Conference on*, pp. 105–113. DOI: 10.1109/DSD.2012.126 (cit. on p. 12).

Schoeberl, Martin, Florain Brandner, Jens Sparsø, and Evangelia Kasapaki (2012). "A Statically Scheduled Time-Division- Multiplexed Network-on-Chip for Real-Time Systems". In: *Sixth IEEE/ACM International Symposium on Networks on Chip (NoCS), 2012*, pp. 152–160. DOI: 10.1109/NOCS.2012.25 (cit. on pp. 47, 61).

Schoeberl, Martin, Florian Brandner, Stefan Hepp, Wolfgang Puffitsch, and Daniel Prokesch (2015). *Patmos Reference Handbook*. Technical University of Denmark. Lyngby, Denmark (cit. on p. 47).

Schoeberl, Martin, Pascal Schleuniger, Wolfgang Puffitsch, Florian Brandner, and Christian W. Probst (2011). "Towards a Time-predictable Dual-Issue Microprocessor: The Patmos Approach". In: *Bringing Theory to Practice: Predictability and Performance in Embedded Systems*. Ed. by Philipp Lucas, Lothar Thiele, Benoit Triquet, Theo Ungerer, and Reinhard Wilhelm. Vol. 18. OpenAccess Series in Informatics (OASIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, pp. 11–21. ISBN: 9783939897286. DOI: http://dx.doi.org/10.4230/OASIcs.PPES.2011.11 (cit. on pp. 32, 47).

Schüpbach, Adrian, Simon Peter, Andrew Baumann, Timothy Roscoe, Paul Barham, Tim Harris, and Rebecca Isaacs (2008). "Embracing diversity in the Barrelfish manycore operating system". In: *Workshop on Managed Many-Core Systems, Boston, MA, USA, June 24, 2008*. ACM (cit. on p. 16).

Sha, Lui, Tarek F. Abdelzaher, Karl-Erik Årzén, Anton Cervin, Theodore P. Baker, Alan Burns, Giorgio C. Buttazzo, Marco Caccamo, John P. Lehoczky, and Aloysius K. Mok (2004). "Real Time Scheduling Theory: A Historical Perspective". In: *Real-Time Systems* 28.2-3, pp. 101–155. DOI: 10.1023/B:TIME.0000045315.61234.1e (cit. on p. 5).

Sha, Lui, Ragunathan Rajkumar, and John P. Lehoczky (1990). "Priority inheritance protocols: an approach to real-time synchronization". In: *Computers, IEEE Transactions on* 39.9, pp. 1175–1185. ISSN: 0018-9340. DOI: 10.1109/12.57058 (cit. on p. 119).

Shin, Kang G. and Xianzhong Cui (1995). "Computing time delay and its effects on real-time control systems". In: *IEEE Transactions on Control Systems Technology* 3.2, pp. 218–224. ISSN: 1063-6536. DOI: 10.1109/87.388130 (cit. on p. 120).

Singhoff, Frank, Jérôme Legrand, Laurent Nana, and Lionel Marcé (2005). "Scheduling and Memory Requirements Analysis with AADL". In: *Proceedings of the 2005 Annual ACM SIGAda International Conference on Ada: The Engineering of Correct and Reliable Software for Real-Time & Distributed Systems Using Ada and Related Technologies*. SigAda '05. Atlanta, GA, USA: ACM, pp. 1–10. ISBN: 1595931856. DOI: 10.1145/1103846.1103847 (cit. on p. 75).

Spuri, Marco and Giorgio C. Buttazzo (1994). "Efficient aperiodic service under earliest deadline scheduling". In: *Real-Time Systems Symposium, 1994., Proceedings.* Pp. 2–11. DOI: 10.1109/REAL.1994.342735 (cit. on p. 123).

— (1996). "Scheduling Aperiodic Tasks in Dynamic Priority Systems". In: *Real-Time Systems* 10.2, pp. 179–210. DOI: 10.1007/BF00360340 (cit. on p. 123).

Stankovic, John A. (2001). "VEST – A Toolset for Constructing and Analyzing Component Based Embedded Systems". English. In: *Embedded Software*. Ed. by Thomas A. Henzinger and Christoph M. Kirsch. Vol. 2211. Lecture Notes in Computer Science. Springer Berlin

Heidelberg, pp. 390–402. ISBN: 9783540426738. DOI: 10.1007/3-540-45449-7_27 (cit. on p. 75).

Stefan, Radu, Anca Molnos, and Kees Goossens (2012). "dAElite: A TDM NoC Supporting QoS, Multicast, and Fast Connection Set-up". In: *IEEE Transactions on Computers* 99.PrePrints. ISSN: 0018-9340. DOI: http://doi.ieeecomputersociety.org/10.1109/TC.2012.117 (cit. on p. 32).

Steinberg, Udo and Bernhard Kauer (2010). "NOVA: A Microhypervisor-based Secure Virtualization Architecture". In: *Proceedings of the 5th European Conference on Computer Systems*. EuroSys '10. Paris, France: ACM, pp. 209–222. ISBN: 9781605585772. DOI: 10.1145/1755913.1755935 (cit. on p. 28).

STMicroelectronics (2013). *UM1472 User Manual – Discovery kit for STM32f407/417 lines*. STMicroelectronics (cit. on p. 160).

Suhendra, Vivz, Tulika Mitra, Abhik Roychoudhury, and Ting Chen (2005). "WCET Centric Data Allocation to Scratchpad Memory". In: *Proc. Real-Time Systems Symposium*, pp. 223–232. DOI: 10.1109/RTSS.2005.45 (cit. on p. 32).

Sutter, Herb (2005). "The Free Lunch Is Over – A Fundamental Turn Toward Concurrency in Software". In: *Dr. Dobb's Journal* 30 (3) (cit. on p. 4).

Swaminathan, S. and G. Manimaran (2002). "A Reliability-Aware Value-Based Scheduler for Dynamic Multiprocessor Real-Time Systems". In: *Proceedings of the 16th International Parallel and Distributed Processing Symposium*. IPDPS '02. Washington, DC, USA: IEEE Computer Society, pp. 39–. ISBN: 0769515738 (cit. on p. 89).

T-CREST (2013). *D 8.2 T-CREST White Paper*. Version 2.0 (cit. on pp. 12, 47, 126).

— (2014). *D 3.8 Integration report of the full system implemented on FPGA*. Version 1.0 (cit. on p. 48).

Templ, Josef (2004). *TDL specification and report*. Tech. rep. T004. Department of Computer Sciences, University of Salzburg (cit. on p. 120).

Texas Instruments (2011). *TMS570LS Series Microcontroller, Technical Reference Manual*. Texas Instruments Incorporated (cit. on p. 4).

— (2014). *Hercules™TMS570 Microcontrollers*. Product Brochure. Texas Instruments Incorporated (cit. on p. 12).

Thiele, Lothar, Samarjit Chakraborty, and Martin Naedele (2000). "Real-time calculus for scheduling hard real-time systems". In: *The 2000 IEEE International Symposium on Circuits and Systems, 2000. Proceedings. ISCAS 2000 Geneva*. Vol. 4, 101–104 vol.4. DOI: 10.1109/ISCAS.2000.858698 (cit. on p. 74).

Tidwell, Terry, Robert Glaubius, Christopher D. Gill, and William D. Smart (2010). "Optimizing Expected Time Utility in Cyber-Physical Systems Schedulers". In: *IEEE 31st Real-Time Systems Symposium (RTSS), 2010*, pp. 193–201. DOI: 10.1109/RTSS.2010.28 (cit. on p. 89).

Tilera (2011). *TILE-Gx8036 Processor Specification Brief*. Tilera Corporation (cit. on pp. 4, 11, 13).

Udayakumaran, Sumesh, Angel Dominguez, and Rajeev Barua (2006). "Dynamic allocation for scratch-pad memory using compile-time decisions". In: *ACM Transactions on Embedded Computing Systems* 5.2, pp. 472–511. ISSN: 1539-9087. DOI: http://doi.acm.org/10.1145/1151074.1151085 (cit. on p. 32).

Ungerer, Theo, Christian Bradatsch, Mike Gerdes, Florian Kluge, Ralf Jahr, Jörg Mische, João. Fernandes, Pavel G. Zaykov, Zlatko Petrov, Bert Boddeker, Sebastian Kehr, Hans Regler, Andreas Hugl, Christine Rochange, Haluk Ozaktas, Hugues Cassé, Armelle Bo-

nenfant, Pascal Sainrat, Ian Broster, Nick Lay, David George, Eduardo Quiñones, Milos Panic, Jaume Abella, Francisco Cazorla, Sascha Uhrig, Mathias Rohde, and Arthur Pyka (2013). "parMERASA – Multi-core Execution of Parallelised Hard Real-Time Applications Supporting Analysability". In: *Digital System Design (DSD), 2013 Euromicro Conference on*, pp. 363–370. DOI: 10.1109/DSD.2013.46 (cit. on pp. 12, 46, 126).

Ungerer, Theo, Francisco Cazorla, Pascal Sainrat, Guillem Bernat, Zlatko Petrov, Christine Rochange, Eduardo Quinones, Mike Gerdes, Marco Paolieri, Julian Wolf, Hugues Cassé, Sascha Uhrig, Irakli Guliashvili, Michael Houston, Florian Kluge, Stefan Metzlaff, and Jörg Mische (2010). "MERASA: Multicore Execution of HRT Applications Supporting Analyzability". In: *IEEE Micro* 30, pp. 66–75. ISSN: 0272-1732. DOI: http://doi.ieeecomputersociety.org/10.1109/MM.2010.78 (cit. on pp. 4, 12).

Vahidi, Ardalan and Azim Eskandarian (2003). "Research advances in intelligent collision avoidance and adaptive cruise control". In: *IEEE Transactions on Intelligent Transportation Systems* 4.3, pp. 143–153. ISSN: 1524-9050. DOI: 10.1109/TITS.2003.821292 (cit. on p. 3).

Vangal, Sriram, Jason Howard, Gregory Ruhl, Saurabh Dighe, Howard Wilson, James Tschanz, David Finan, Priya Iyer, Arvind Singh, Tiju Jacob, Shailendra Jain, Sriram Venkataraman, Yatin Hoskote, and Nitin Borkar (2007). "An 80-Tile 1.28TFLOPS Network-on-Chip in 65nm CMOS". In: *IEEE International Solid-State Circuits Conference, 2007. ISSCC 2007. Digest of Technical Papers.* Pp. 98–589. DOI: 10.1109/ISSCC.2007.373606 (cit. on p. 4).

Villalpando, Carlos Y., Andrew E. Johnson, Raphael Some, Jacob Oberlin, and Steven Goldberg (2010). "Investigation of the Tilera processor for real time hazard detection and avoidance on the Altair Lunar Lander". In: *2010 IEEE Aerospace Conference*, pp. 1–9. DOI: 10.1109/AERO.2010.5447023 (cit. on p. 12).

Villalpando, Carlos, David Rennels, Raphael Some, and Manuel Cabanas-Holmen (2011). "Reliable multicore processors for NASA space missions". In: *2011 IEEE Aerospace Conference*, pp. 1–12. DOI: 10.1109/AERO.2011.5747447 (cit. on p. 12).

Wang, Jinggang and Binoy Ravindran (2004). "Time-utility function-driven switched Ethernet: packet scheduling algorithm, implementation, and feasibility analysis". In: *IEEE Transactions on Parallel and Distributed Systems* 15.2, pp. 119–133. ISSN: 1045-9219. DOI: 10.1109/TPDS.2004.1264796 (cit. on p. 89).

Wang, Xi, Tao Tang, and Lianchuan Ma (2012). "A fail-safe infrastructure designed for COTS component used in safety critical system". In: *Signal Processing (ICSP), 2012 IEEE 11th International Conference on.* Vol. 3, pp. 2208–2211. DOI: 10.1109/ICoSP.2012.6492019 (cit. on p. 12).

Wehmeyer, Lars and Peter Marwedel (2005). "Influence of Memory Hierarchies on Predictability for Time Constrained Embedded Software". In: *Proc. Design, Automation and Test in Europe (DATE)*, pp. 600–605. DOI: http://doi.ieeecomputersociety.org/10.1109/DATE.2005.183 (cit. on p. 32).

Wentzlaff, David and Anant Agarwal (2009). "Factored operating systems (fos): the case for a scalable operating system for multicores". In: *SIGOPS Oper. Syst. Rev.* 43 (2), pp. 76–85. ISSN: 0163-5980. DOI: 10.1145/1531793.1531805 (cit. on pp. 18, 35, 126).

Wentzlaff, David, Charles Gruenwald III, Nathan Beckmann, Kevin Modzelewski, Adam Belay, Lamia Youseff, Jason Miller, and Anant Agarwal (2010). "A Unified Operating System for Clouds and Manycore: fos". In: *1st Workshop on Computer Architecture and Operating System co-design (CAOS), Jan 2010* (cit. on p. 18).

Wentzlaff, David, Charles Gruenwald III, Nathan Beckmann, Kevin Modzelewski, Adam Belay, Lamia Youseff, Jason Miller, and Anant Agarwal (2010). "An operating system for multicore and clouds: mechanisms and implementation". In: *1st ACM symposium on Cloud computing, Indianapolis, Indiana, USA*. SoCC '10. New York, NY, USA: ACM, pp. 3–14. ISBN: 9781450300360. DOI: 10.1145/1807128.1807132 (cit. on p. 19).

Wentzlaff, David, III Gruenwald Charles, Nathan Beckmann, Adam Belay, Harshad Kasture, Kevin Modzelewski, Lamia Youseff, Jason E. Miller, and Anant Agarwal (2011). *Fleets: Scalable Services in a Factored Operating System*. Tech. rep. MIT-CSAIL-TR-2011-012. Massachusetts Institute of Technology (cit. on p. 19).

Wermund, Wolfgang (2015). "Untersuchung der zeitlichen Vorhersagbarkeit des eMesh-NoCs des Epiphany Manycore-Prozessors". Bachelor thesis. Universität Augsburg (cit. on p. 124).

West, Richard and Karsten Schwan (1999). "Dynamic window-constrained scheduling for multimedia applications". In: *IEEE International Conference on Multimedia Computing and Systems, 1999.* Vol. 2, 87–91 vol.2. DOI: 10.1109/MMCS.1999.778145 (cit. on p. 87).

Wilhelm, Reinhard, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström (2008). "The Worst-case Execution Time Problem—Overview of Methods and Survey of Tools". In: *ACM Transactions on Embedded Computing Systems (TECS)* 7.3. DOI: 10.1145/1347375.1347389 (cit. on p. 9).

Wilhelm, Reinhard, Daniel Grund, Jan Reineke, Marc Schlickling, Markus Pister, and Christian Ferdinand (2009). "Memory Hierarchies, Pipelines, and Buses for Future Architectures in Time-Critical Embedded Systems". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 28.7, pp. 966–978. ISSN: 0278-0070. DOI: 10.1109/TCAD.2009.2013287 (cit. on pp. 4, 13).

Wittenmark, Björn, Johan Nilsson, and Martin Törngren (1995). "Timing problems in real-time control systems". In: *American Control Conference, Proceedings of the 1995*. Vol. 3, 2000–2004 vol.3. DOI: 10.1109/ACC.1995.531240 (cit. on p. 120).

Wu, Haisang, Binoy Ravindran, E. Douglas Jensen, and Umut Balli (2004). "Utility accrual scheduling under arbitrary time/utility functions and multiunit resource constraints". In: *in IEEE Real-Time and Embedded Computing Systems and Applications*, pp. 80–98 (cit. on p. 89).

Xi, Sisu, Justin Wilson, Chenyang Lu, and Christopher Gill (2011). "RT-Xen: Towards real-time hypervisor scheduling in Xen". In: *Proceedings of the International Conference on Embedded Software (EMSOFT), 2011*, pp. 39–48 (cit. on p. 28).

# Acronyms

**AC** actuator component

**ATS** abstract task set

**FI** full image

**BCET** best-case execution time

**BET** bounded execution time

**BSW** basic software

**CBS** constant bandwidth server

**CC** computation component

**CMA** Chen and Muhlethaler's algorithm

**COTS** commercial off the shelve

**CPS** cyber-physical system

**CTS** concrete task set

**DASA** dependent activities scheduling algorithm

**DBP** distance-based priority

**DMA** direct memory access

**DVD** dynamic value density

**ECU** electronic control unit

**EDF** earliest deadline first

**EM** execution model

**EMS** engine management system

**EMU** engine management unit

**FA** federated architecture

**FIFO** first-in first-out

**FPGA** field programmable gate array

**GDPA** Guaranteed Dynamic Priority Assignment

**GDPA-S** Simplified Guaranteed Dynamic Priority Assignment

**GST** Goossens' schedulability test

**GTM** generic timing model

**GUI** graphical user interface

**GUS** generic utility scheduling

**HAL** hardware abstraction layer

**HCUF** history-cognisant utility function

**IMA** integrated modular avionics

**IPC** inter process communication

**IPCS** inter-partition communication server

**IPO** input-process-output

**IRQ** interrupt request

**ISA** instruction set architecture

**ISR** interrupt service routine

**LBESA** Locke's Best-Effort Scheduling Algorithm

**LET** logical execution time

**MCAL** MicroController Abstraction Layer

**MMU** memory management unit

**MOSSCA** manycore operating system for safety-critical applications

**NoC** network-on-chip

**NUMA** non-uniform memory access

**OS** operating system

**QoS** quality of service

**RPC** remote procedure call

**RTE** run-time environment

**RTES** real-time embedded system

**RTI** real-time interrupt

**RTOS** real-time operating system

**RUA** resource-constrained utility accrual

**SC** sensor component

**SCCT** simple capture/compare timer

**SCS** safety-critical system

**SD** self-distributing kernel

**SI** split image

**SPM** scratchpad memory

**SRM** synchronous reactive model

**TBS** total bandwidth server

**TLB** translation look-aside buffer

**TUF** time-utility function

**UA** utility-accrual

**UAV** unmanned aerial vehicle

**VMM** virtual machine monitor

**WCD** worst-case duration

**WCET** worst-case execution time

**WCRT** worst-case response time

**WCTT** worst-case traversal time

**WCWT** worst-case waiting time

**ZET** zero execution time

# Appendices

# A Further Results on $(m, k)$-Firm Real-Time Tasks

This appendix contains additional performance results of the evaluations described in chapter 8. The displayed numbers are found during the simulation of task sets with restricted $m$ parameter (see sect. 8.5.3), and restricted $m$ parameter and realistic periods (see sect. 8.5.3).

## A.1. Restricted $m$ Parameter

The diagram for $r_{\mathrm{m}} = 0.5$ is omitted, as it is already shown in figure 8.9. All other results from section 8.5.3 are displayed in figures A.1 to A.8. The legends are the same for all figures. They are omitted in figures A.5 to A.8 for better presentability.

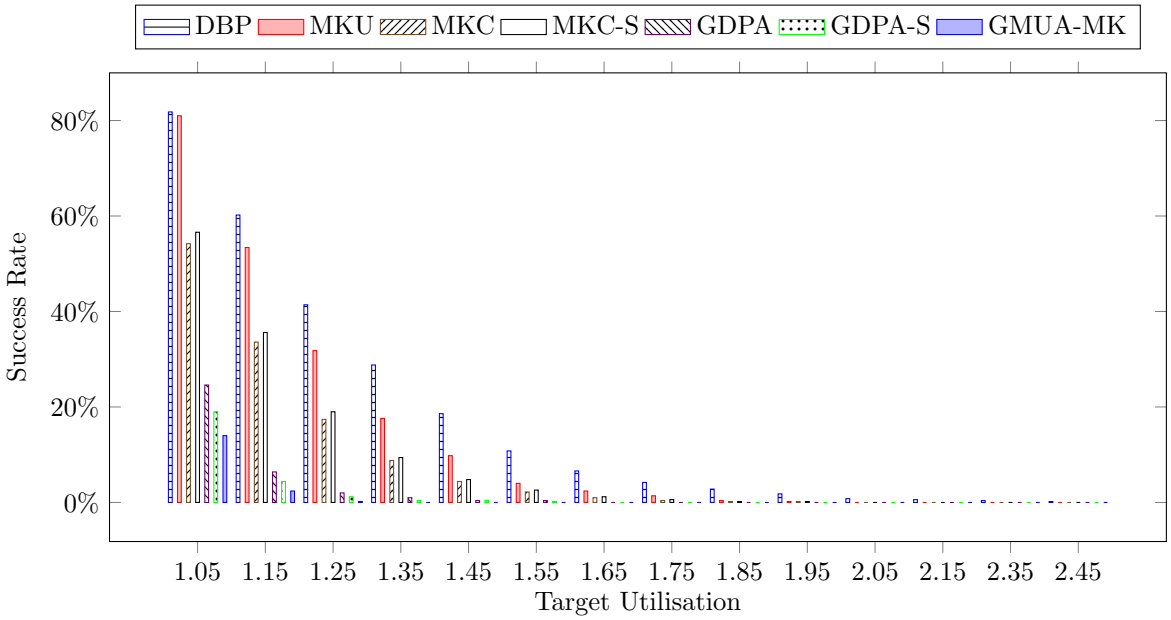



Figure A.1.: Performance with restricted $m_i$, $r_{\mathrm{m}} = 0.1$

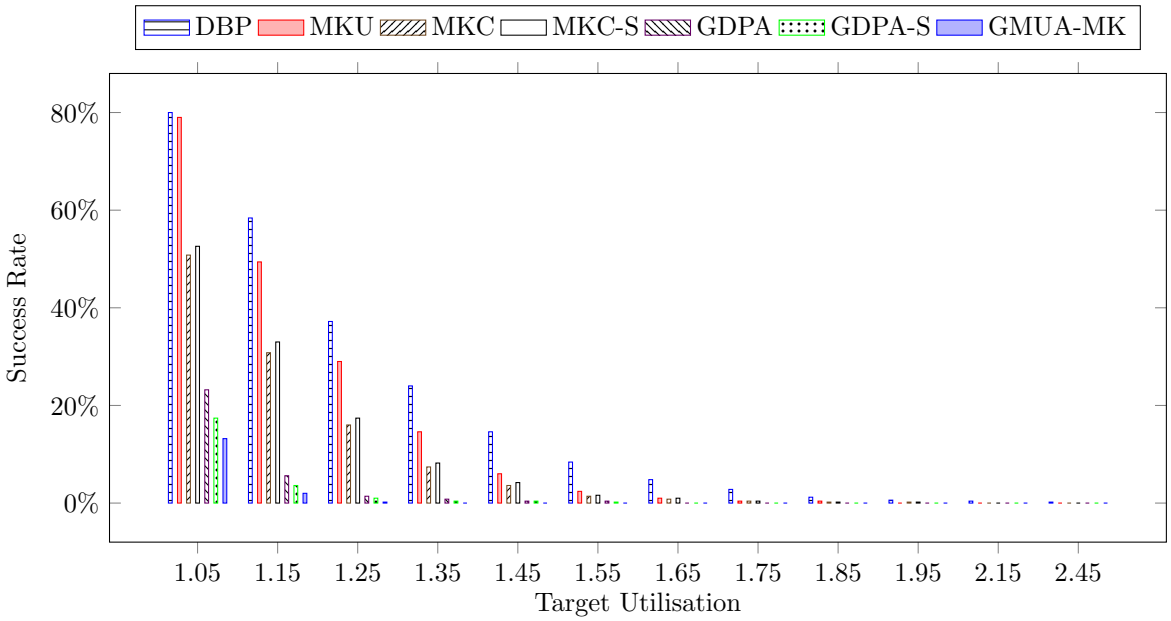



Figure A.2.: Performance with restricted $m_i$, $r_{\mathrm{m}} = 0.2$

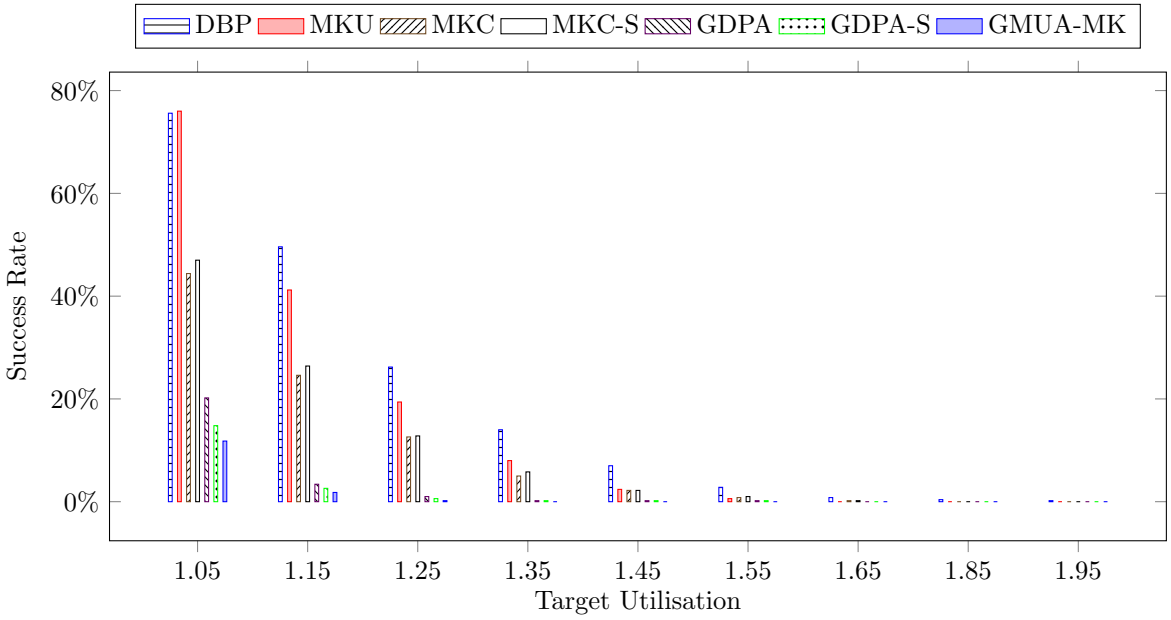



Figure A.3.: Performance with restricted $m_i$, $r_{\mathrm{m}} = 0.3$

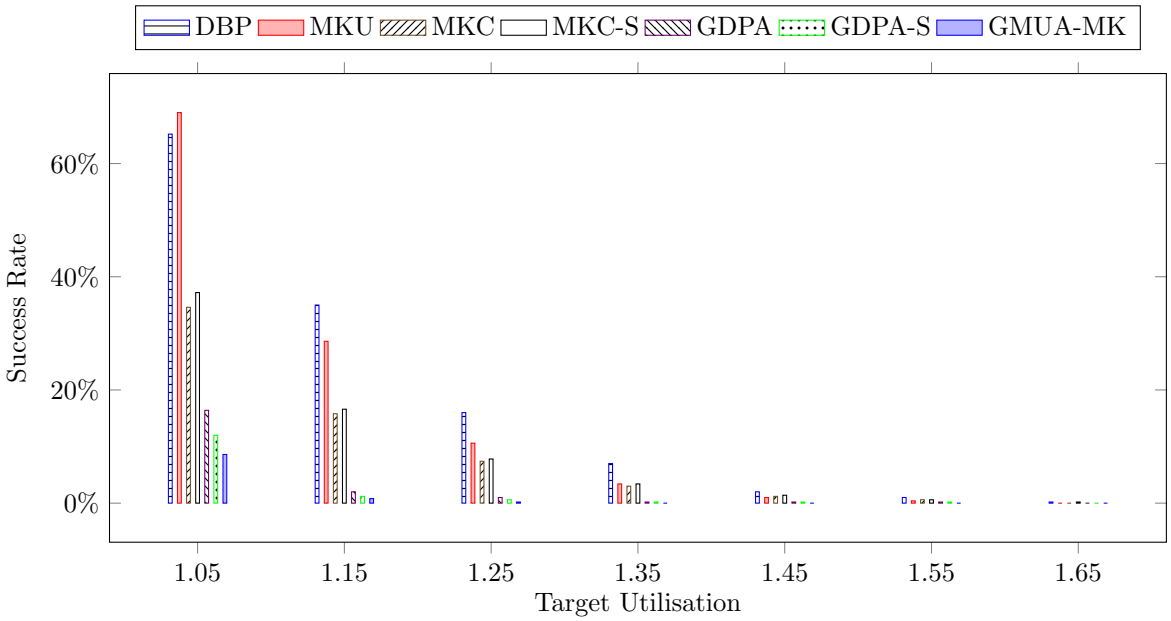



Figure A.4.: Performance with restricted $m_i$, $r_{\mathrm{m}} = 0.4$

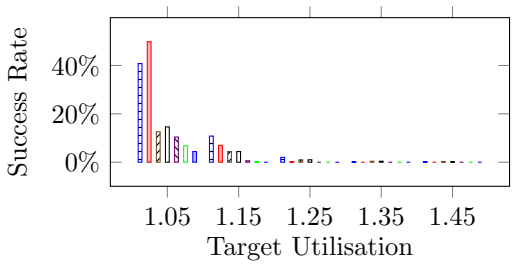



Figure A.5.: Performance with restricted $m_i$, $r_{\mathrm{m}} = 0.6$

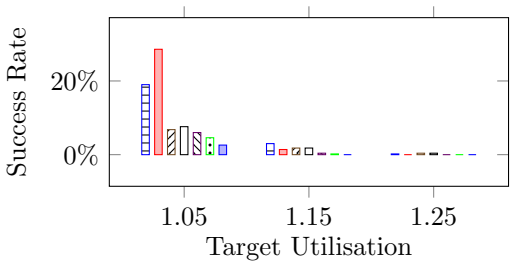



Figure A.6.: Performance with restricted $m_i$, $r_{\mathrm{m}} = 0.7$

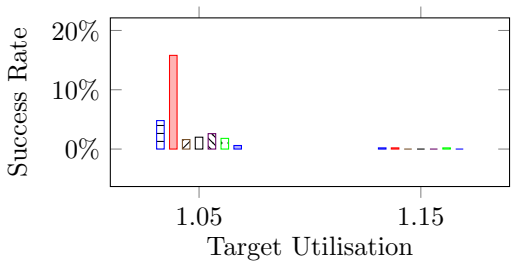



Figure A.7.: Performance with restricted $m_i$, $r_{\mathrm{m}} = 0.8$

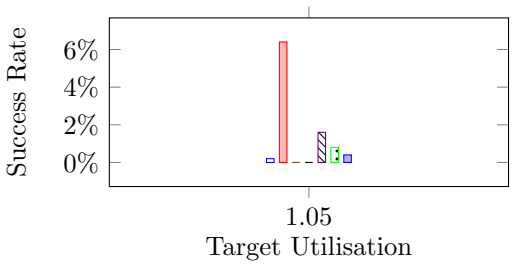



Figure A.8.: Performance with restricted $m_i$, $r_{\mathrm{m}} = 0.9$

## A.2. Restricted $m$ Parameter and Realistic Periods

Figures A.9 to A.13 show the results from the evaluations discussed in section 8.5.3. Again, legends in figures A.10 to A.13 are omitted for better presentability, please refer to figure A.9.

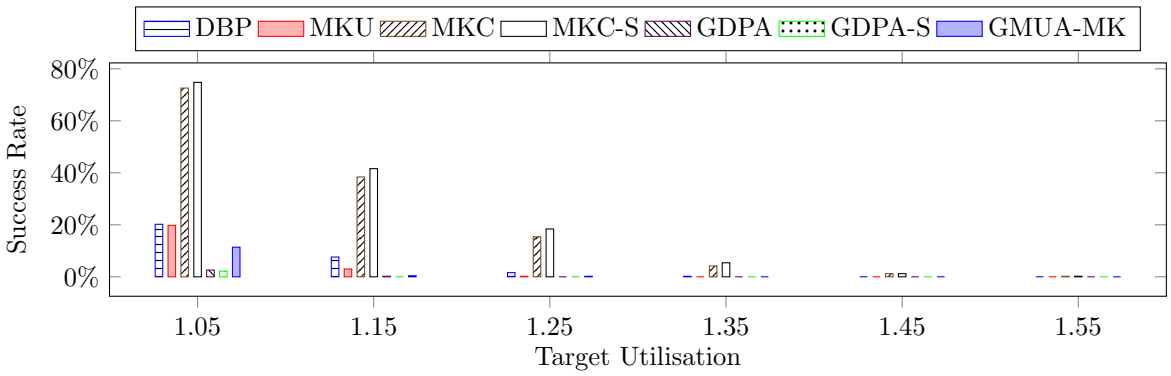



Figure A.9.: Performance with restricted $m_i$ ($r_\mathrm{m} = 0.5$) and realistic periods

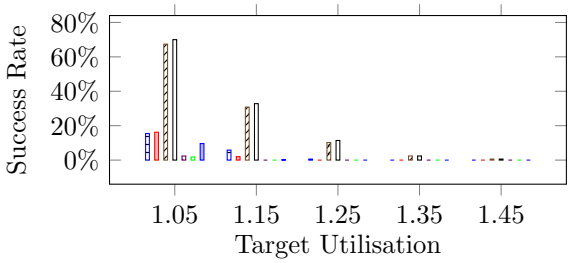



Figure A.10.: Performance with restricted $m_i$ ($r_\mathrm{m} = 0.6$) and realistic periods

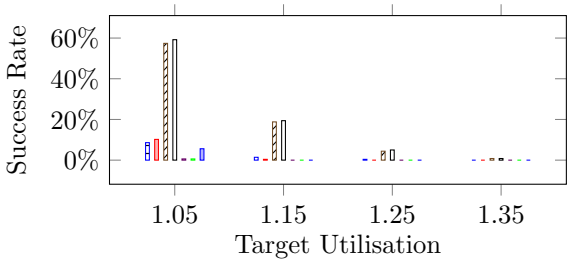



Figure A.11.: Performance with restricted $m_i$ ($r_\mathrm{m} = 0.7$) and realistic periods

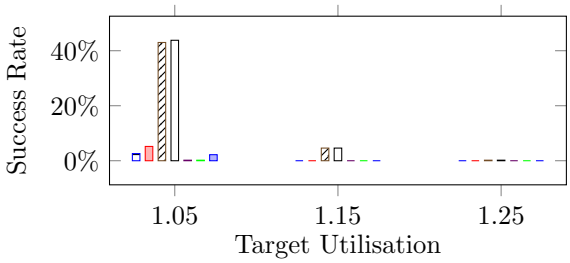



Figure A.12.: Performance with restricted $m_i$ ($r_\mathrm{m} = 0.8$) and realistic periods

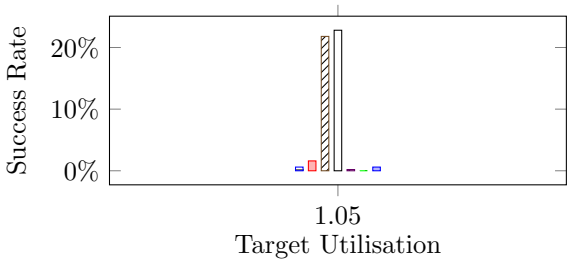



Figure A.13.: Performance with restricted $m_i$ ($r_\mathrm{m} = 0.9$) and realistic periods

# B Tools

In the course of this work, a number of tools was developed and released as open source software. This appendix provides a brief overview over these tools. Section B.1 describes `mwsim`, which is used for simulation of worst-case paths in manycore programs. Details about the scheduling simulator `tms-sim` can be found in section B.2. The benchmark system EMSBench is described in section B.3. Related to EMSBench, the IP-core SCCT was developed, which is described in section B.4.

## B.1. MWSim

The `mwsim` tool is used to calculate worst-case durations of processes in manycore processors. Details about its functionality are described in sect. 6.2.2 and Kluge, Gerdes, and Ungerer 2014b.

## B.2. `tms-sim`

The *Timing Models Scheduling SIMulator (`tms-sim`)* was originally developed to evaluate the performance of time-utility-based schedulers (Kluge et al. 2013). The works on task sets with relaxed real-time constraints (see ch. 8) have led to a number of extensions of `tms-sim`. In the aftermath of the parMERASA project, it was also used to evaluate the performance of different execution models (Bradatsch et al. 2016). Here, the focus lies on the age of data that is exchanged between tasks. The internal functionality of `tms-sim` is detailed in Kluge 2014. `tms-sim` is available as open source software[1].

## B.3. EMSBench

The EMSBench package is motivated by the fact that only few *system benchmarks* for embedded systems exist. Typical benchmark suites consist of small, isolated programs. These do not exhibit the complex behaviour of contemporary RTES. As a central contribution to this problem, EMSBench (Kluge and Ungerer 2015) provides two programs. EMSBench.ems is based on the open source EMS FreeEMS, where most input signals are removed. Only

[1] `http://myweb.rz.uni-augsburg.de/~klugeflo/tms-sim/`

inputs from crank resp. cam shaft sensor are kept as they have most influence of the program's internal behaviour. EMSBench.tg is a trace generator that provides crank/cam shaft signal traces based on arbitrary driving cycles. A HAL ensures that the embedded parts of EMSBench (EMSBench.ems and EMSBench.tg) can be ported to arbitrary platforms. So far, implementations for the STM32F4-Discovery platform (STMicroelectronics 2013) from STMicroelectronics and a custom Nios II FGPA platform are available. Current works with EMSBench deal with its use as benchmark for static WCET analysis. EMSBench is available as open source software[2].

## B.4. SCCT

During the development of EMSBench, a HAL was defined to ensure portability of the code. While initially implemented on the STM32F4-Discovery platform, a port to a custom Nios II platform was developed to ensure completeness of the HAL. Nios II is an IP-core developed by Altera[3] that can be deployed to field programmable gate arrays (FPGAs). Several peripheral IP-cores are also available from Altera. As, at the moment of this work, no IP-core with a capture/compare timer was available, an own one was developed, the *simple capture/compare timer (SCCT)* (Kluge 2015). SCCT features a global counter register with prescaler. The number of capture/compare channels can be configured before compilation. SCCT was released as open source software under the GNU General Public License and can be downloaded from the OpenCores website[4].

[2] `https://github.com/unia-sik/emsbench`
[3] `https://www.altera.com/`
[4] `http://opencores.org/project,scct`

# C Publications

## C.1. Related to this Work (Peer-Reviewed)


Kluge, Florian, Markus Neuerburg, and Theo Ungerer (2015). "Utility-based scheduling of $(m, k)$-Firm Real-Time Task Sets". In: *Accepted at 28th GI/ITG International Conference on Architecture of Computing Systems (ARCS 2015)*.

Kluge, Florian and Theo Ungerer (2015). "EMSBench: Benchmark und Testumgebung für reaktive Systeme". In: *Betriebssysteme und Echtzeit, Echtzeit 2015, Fachtagung des gemeinsamen Fachausschusses Echtzeitsysteme von Gesellschaft für Informatik e.V. (GI), VDI/VDE-Gesellschaft für Mess- und Automatisierungstechnik (GMA) und Informationstechnischer Gesellschaft im VDE (ITG) sowie der Fachgruppe Betriebssysteme von GI und ITG, Boppard, Germany, 12.-13. November 2015*. Ed. by Wolfgang A. Halang and Olaf Spinczyk. Informatik Aktuell. Springer, pp. 11–20. DOI: 10.1007/978-3-662-48611-5_2.

Kluge, Florian, Mike Gerdes, Florian Haas, and Theo Ungerer (2014). "A Generic Timing Model for Cyber-Physical Systems". In: *Workshop Reconciling Performance and Predictability (RePP'14)*. Grenoble, France. DOI: 10.13140/2.1.1820.4165.

Kluge, Florian, Mike Gerdes, and Theo Ungerer (2014a). "An Operating System for Safety-Critical Applications on Manycore Processors". In: *17th IEEE International Symposium on Object Oriented Real-Time Distributed Computing (ISORC), 2014*. Reno, NV, USA: IEEE, pp. 238–245.

— (2014b). "The Boot Process in Real-time Manycore Processors". In: *Proceedings of the 22nd International Conference on Real-Time Networks and Systems*. RTNS '14. Versaille, France: ACM, 77:77–77:86. ISBN: 9781450327275. DOI: 10.1145/2659787.2659816.

Bradatsch, Christian, Florian Kluge, and Theo Ungerer (2013). "A Cross-Domain System Architecture for Embedded Hard Real-Time Many-Core Systems". In: *High Performance Computing and Communications 2013 IEEE International Conference on Embedded and Ubiquitous Computing (HPCC/EUC), 2013 IEEE 10th International Conference on*, pp. 2034–2041. DOI: 10.1109/HPCC.and.EUC.2013.293.

Kluge, Florian, Florian Haas, Mike Gerdes, and Theo Ungerer (2013). "History-Cognisant Time-Utility-Functions for Scheduling Overloaded Real-Time Control Systems". In: *Proceedings of 7th Junior Researcher Workshop on Real-Time Computing (JRWRTC 2013)*. Sophia Antipolis, France. DOI: 10.13140/2.1.1080.0648.

Gerdes, Mike, Florian Kluge, Theo Ungerer, Christine Rochange, and Pascal Sainrat (2012). "Time Analysable Synchronisation Techniques for Parallelised Hard Real-Time Applications". In: *Proc. Design, Automation and Testing in Europe (DATE 2012)*, pp. 671–676.

Kluge, Florian, Mike Gerdes, and Theo Ungerer (2012). "AUTOSAR OS on a Message-Passing Multicore Processor". In: *Proceedings of 7th IEEE International Symposium on Industrial Embedded Systems (SIES'12) (Posters)*. Karlsruhe, Germany.

Kluge, Florian, Benoît Triquet, Christine Rochange, and Theo Ungerer (2012). "Operating Systems for Manycore Processors from the Perspective of Safety-Critical Systems". In: *Proceedings of 8th annual workshop on Operating Systems for Embedded Real-Time applications (OSPERT 2012)*. Pisa, Italy. DOI: 10.13140/2.1.1211.1362.

## C.2. Further Peer-Reviewed Publications

Bradatsch, Christian, Florian Kluge, and Theo Ungerer (2016). "Data Age Diminution in the Logical Execution Time Model". In: *29th GI/ITG International Conference on Architecture of Computing Systems (ARCS 2016), Nuremberg, Germany, April 4-7, 2016, Proceedings*. Accepted for publication.

— (2014a). "Comparison of Service Call Implementations in an AUTOSAR Multi-core OS". In: *9th IEEE International Symposium on Industrial Embedded Systems (SIES 2014)*. Pisa, Italy.

— (2014b). "Synchronous Execution of a Parallelised Interrupt Handler". In: *20th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2014), Work-in-Progress Session*, pp. 9–10.

Roth, Michael, Julia Schmitt, Florian Kluge, and Theo Ungerer (2013). "DHT Broadcast Optimisation with ID Assignment Rules." In: *ESOS*.

Ungerer, Theo, Christian Bradatsch, Mike Gerdes, Florian Kluge, Ralf Jahr, Jörg Mische, João. Fernandes, Pavel G. Zaykov, Zlatko Petrov, Bert Boddeker, Sebastian Kehr, Hans Regler, Andreas Hugl, Christine Rochange, Haluk Ozaktas, Hugues Cassé, Armelle Bonenfant, Pascal Sainrat, Ian Broster, Nick Lay, David George, Eduardo Quiñones, Milos Panic, Jaume Abella, Francisco Cazorla, Sascha Uhrig, Mathias Rohde, and Arthur Pyka (2013). "parMERASA – Multi-core Execution of Parallelised Hard Real-Time Applications Supporting Analysability". In: *Digital System Design (DSD), 2013 Euromicro Conference on*, pp. 363–370. DOI: 10.1109/DSD.2013.46.

Gerdes, Mike, Florian Kluge, Christine Rochange, and Theo Ungerer (2012). "The Split-Phase Synchronisation Technique: Reducing the Pessimism in the WCET Analysis of Parallelised Hard Real-Time Programs". In: *18th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), Seoul, Korea*. IEEE, pp. 88–97. DOI: 10.1109/RTCSA.2012.11.

Kiefhaber, Rolf, Stephan Hammer, Benjamin Savs, Julia Schmitt, Michael Roth, Florian Kluge, Elisabeth André, and Theo Ungerer (2011). "The Neighbor-Trust Metric to Measure Reputation in Organic Computing Systems". In: *Proceedings of the 2nd Workshop on Trustworthy Self-Organizing Systems (TSOS 2011)*.

Schmitt, Julia, Michael Roth, Rolf Kiefhaber, Florian Kluge, and Theo Ungerer (2011a). "Concept of a Reflex Manager to Enhance the Planner Component of an Autonomic / Organic System". In: *Autonomic and Trusted Computing*. Vol. 6906. Lecture Notes in Computer Science, pp. 19–30. ISBN: 9783642234958.

— (2011b). "Realizing Self-x Properties by an Automated Planner". In: *Proceedings of the 8th ACM international conference on Autonomic computing*. ICAC 2011. Karlsruhe, Germany, pp. 185–186. ISBN: 9781450306072.

— (2011c). "Using an Automated Planner to Control an Organic Middleware". In: *Fifth International Conference on Self-Adaptive and Self-Organizing Systems (SASO 2011)*, pp. 71–78.

Wolf, Julian, Mike Gerdes, Florian Kluge, Sascha Uhrig, Jörg Mische, Stefan Metzlaff, Christine Rochange, Hugues Cassé, Pascal Sainrat, and Theo Ungerer (2011). "RTOS support for execution of parallelized hard real-time tasks on the MERASA multi-core processor". In: *Comput. Syst. Sci. Eng.* 26.6.

Kluge, Florian, Sascha Uhrig, Jörg Mische, Benjamin Satzger, and Theo Ungerer (2010a). "Dynamic Workload Prediction for Soft Real-Time Applications". In: *7th IEEE International Conferences on Embedded Software and Systems (ICESS-10), Bradford, United Kingdom, June 29 - July 1, 2010, Proceedings*. Bradford, UK.

— (2010b). "Optimisation of Energy Consumption of Soft Real-Time Applications by Workload Prediction". In: *First IEEE Workshop on Self-Organizing Real-Time Systems (SORT 2010), Carmona, Spain, May 4, 2010, Proceedings*. Carmona, Spain, pp. 63–72.

Satzger, Benjamin, Faruk Bagci, Florian Kluge, and Theo Ungerer (2010). "Towards lightweight self-configuration in wireless sensor networks". In: *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC), Sierre, Switzerland, March 22-26, 2010*, pp. 791–792.

Ungerer, Theo, Francisco Cazorla, Pascal Sainrat, Guillem Bernat, Zlatko Petrov, Christine Rochange, Eduardo Quinones, Mike Gerdes, Marco Paolieri, Julian Wolf, Hugues Cassé, Sascha Uhrig, Irakli Guliashvili, Michael Houston, Florian Kluge, Stefan Metzlaff, and Jörg Mische (2010). "MERASA: Multicore Execution of HRT Applications Supporting Analyzability". In: *IEEE Micro* 30, pp. 66–75. ISSN: 0272-1732. DOI: `http://doi.ieeecomputersociety.org/10.1109/MM.2010.78`.

Wolf, Julian, Mike Gerdes, Florian Kluge, Sascha Uhrig, Jörg Mische, Stefan Metzlaff, Christine Rochange, Hugues Cassé, Pascal Sainrat, and Theo Ungerer (2010). "RTOS Support for Parallel Execution of Hard Real-Time Applications on the MERASA Multi-core Processor". In: *Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2010 13th IEEE International Symposium on*, pp. 193–201. DOI: `10.1109/ISORC.2010.31`.

Bagci, Faruk, Florian Kluge, Benjamin Satzger, and Theo Ungerer (2009). "Towards Indoor Location Estimation and Tracking with Wireless Sensors". In: *IEEE International Symposium on Intelligent Signal Processing (WISP 09), Budapest, Hungary, August 26-28, 2009*, pp. 235–240.

Bagci, Faruk, Florian Kluge, Theo Ungerer, and Nader Bagherzadeh (2009). "Optimisations for LocSens – an indoor location tracking system using wireless sensors". In: *Intational Journal of Sensor Networks* 6.3/4, pp. 157–166. ISSN: 1748-1279. DOI: `http://dx.doi.org/10.1504/IJSNET.2009.029392`.

Kluge, Florian, Chenglong Yu, Jörg Mische, Sascha Uhrig, and Theo Ungerer (2009). "Implementing AUTOSAR Scheduling and Resource Management on an Embedded SMT

Processor". In: *12th International Workshop on Software and Compilers for Embedded Systems (SCOPES 2009)*. Nice, France, pp. 33–41.

Mische, Jörg, Sascha Uhrig, Florian Kluge, and Theo Ungerer (2009). "IPC Control for Multiple Real-Time Threads on an In-order SMT Processor". In: *Proceedings of the 4th International Conference on High Performance and Embedded Architectures and Compilers (HiPEAC 2009)*. Paphos, Cyprus, pp. 125–139.

Satzger, Benjamin, Florian Mutschelknaus, Faruk Bagci, Florian Kluge, and Theo Ungerer (2009). "Towards Trustworthy Self-optimization for Distributed Systems". In: *Software Technologies for Embedded and Ubiquitous Systems, 7th IFIP WG 10.2 International Workshop, SEUS 2009, Newport Beach, CA, USA, November 16-18, 2009, Proceedings*, pp. 58–68.

Bagci, Faruk, Florian Kluge, Nader Bagherzadeh, and Theo Ungerer (2008). "LocSens - An Indoor Location Tracking System using Wireless Sensors". In: *Proceedings of the 17th International Conference on Computer Communications and Networks, IEEE ICCCN 2008, St. Thomas, U.S. Virgin Islands, August 3-7, 2008*, pp. 887–891.

Kluge, Florian, Jörg Mische, Sascha Uhrig, and Theo Ungerer (2008a). "An Operating System Architecture for Organic Computing in Embedded Real-Time Systems". In: *Proceedings of the 5th International Conference on Autonomic and Trusted Computing (ATC 2008)*. Oslo, Norway: Springer, pp. 343–357.

— (2008b). "Building Adaptive Embedded Systems by Monitoring and Dynamic Loading of Application Modules". In: *Workshop on Adaptive and Reconfigurable Embedded Systems (APRES'08)*. St. Louis, MO, USA, pp. 23–26.

Kluge, Florian, Sascha Uhrig, Jörg Mische, and Theo Ungerer (2008). "A Two-Layered Management Architecture for Building Adaptive Real-time Systems". In: *Proceedings of The 6th IFIP Workshop on Software Technologies for Future Embedded & Ubiquitous Systems (SEUS 2008)*. Capri Island, Italy: Springer, pp. 126–137.

Mische, Jörg, Sascha Uhrig, Florian Kluge, and Theo Ungerer (2008). "Exploiting Spare Resources of In-order SMT Processors Executing Hard Real-time Threads". In: *IEEE International Conference on Computer Design 2008 (ICCD 08)*. Lake Tahoe, CA, USA, pp. 371–376.

Kluge, Florian, Jörg Mische, Sascha Uhrig, Theo Ungerer, and Rafael Zalman (2007). "Use of Helper Threads for OS Support in the Multithreaded Embedded TriCore 2 Processor". In: *Proceedings Work-In-Progress-Session of the 13th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2007)*. Ed. by Chenyang Lu. Bellevue, WA, USA, pp. 25–27.

## C.3. Book Chapters

Kluge, Florian, Jörg Mische, and Theo Ungerer (2011). "Dynamic Classification for Embedded Real-Time Systems". In: *Organic Computing – A Pardigm Shift for Complex Systems*. Ed. by Theo Ungerer Christian Müller-Schloer Hartmut Schmeck. Springer, pp. 579–581.

Roth, Michael, Julia Schmitt, Rolf Kiefhaber, Florian Kluge, and Theo Ungerer (2011). "Organic Computing Middleware for Ubiquitous Environments". In: *Organic Computing - A Paradigm Shift for Complex Systems*. Springer Basel, pp. 339–351.

Bagci, Faruk, Florian Kluge, Benjamin Satzger, Andreas Pietzowski, Wolfgang Trumler, and Theo Ungerer (2008). "Experiences with a Smart Office Project". In: *Mobile Intelligence: Mobile Computing and Computational Intelligence*. Ed. by Laurence T. Yang. Wiley-Interscience. ISBN: 9780470195550.

## C.4. Technical Reports

Kluge, Florian (2016). *Notes on the Generation of Spin-Values for Fixed $(m, k)$-Patterns*. Tech. rep. 2016-01. Department of Computer Science, University of Augsburg.

— (2015). *A Simple Capture/Compare Timer*. Tech. rep. 2015-01. Department of Computer Science, University of Augsburg. DOI: 10.13140/2.1.1251.2321.

— (2014). *tms-sim – Timing Models Scheduling Simulation Framework – Release 2014-12*. Tech. rep. 2014-07. Department of Computer Science, University of Augsburg. DOI: 10.13140/2.1.1251.2321.

Kluge, Florian and Julian Wolf (2008). *Basic System-Level Software for a Single-Core MERASA Processor*. Tech. rep. 2008-06. Department of Computer Science, University of Augsburg.

## C.5. Further Publications (Without Review)

Kluge, Florian (2008). *Ortsbestimmung in Bürogebäuden*. VDM Verlag Dr. Müller. ISBN: 987-3-639-06380-6.

Kluge, Florian, Jörg Mische, Stefan Metzlaff, Sascha Uhrig, and Theo Ungerer (2007). "Integration of Hard Real-Time and Organic Computing". In: *ACACES 2007 Poster Abstracts*. L'Aquila, Italy: Academia Press, Ghent (Belgium), pp. 295–298.

Kluge, Florian, Jörg Mische, Sascha Uhrig, and Theo Ungerer (2006). "CAR-SoC - Towards an Autonomic SoC Node". In: *ACACES 2006 Poster Abstracts*. L'Aquila, Italy: Academia Press, Ghent (Belgium), pp. 37–40.

## C.6. Theses

Kluge, Florian (2010). "Autonomic- und Organic-Computing-Techniken in eingebetteten Echtzeitsystemen". Dissertation. Universität Augsburg.

— (2005). "Untersuchung bestehender Methoden und Entwurf eines Systems zur Ortsbestimmung in Bürogebäuden". Diplomarbeit. Universität Augsburg.